高等学校计算机基础教育教材精选

大学计算机

——计算、构造与设计（第2版）

吴宁　主编
崔舒宁　夏秦　编著

清华大学出版社

高等学校计算机基础教育教材精选

大学计算机

——计算、构造与设计（第2版）

吴　宁　主编
崔舒宁　夏　秦　编著

清华大学出版社
北京

内 容 简 介

本书为“大学计算机”国家精品资源共享课程专用教材和 MOOC 课程参考教材。全书以“计算思维能力”培养为出发点,围绕计算、构造、设计三大主题进行内容组织,将核心聚焦到计算模型与信息编码、系统构造与抽象、算法与数据结构设计三大模块,强调自底向上的构造思维能力、逻辑分析能力与编程实现能力。

全书共 8 章,包括计算机与计算机科学引论,信息表示与编码,系统软硬件构造,网络应用及网络安全技术,C 语言程序设计基础,数组、函数和指针,算法分析与设计,数据结构基础。作为 MOOC 课程参考教材,本书主体内容配有教学微视频及包括动画演示案例、在线作业练习等各类辅助教学和学习的网络数字资源。

本书可作为普通高等学校理工科各类专业学生学习“大学计算机基础”课程的教材,适用学时为 48~64 学时。书(目录)中带有 * 的章节为选讲内容,可根据情况课内讲授或作为翻转课堂教学使用。

图书在版编目(CIP)数据

大学计算机:计算、构造与设计/吴宁主编. --2 版. --北京:清华大学出版社,2016(2018.1 重印)
高等学校计算机基础教育教材精选
ISBN 978-7-302-44599-9

Ⅰ. ①大…　Ⅱ. ①吴…　Ⅲ. ①电子计算机—高等学校—教材　Ⅳ. ①TP3

中国版本图书馆 CIP 数据核字(2016)第 175381 号

责任编辑:焦　虹　战晓雷
封面设计:常雪影
责任校对:时翠兰
责任印制:王静怡

出版发行:清华大学出版社
网　　址:http://www.tup.com.cn, http://www.wqbook.com
地　　址:北京清华大学学研大厦 A 座　**邮　　编**:100084
社 总 机:010-62770175　**邮　　购**:010-62786544
投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn
课件下载:http://www.tup.com.cn,010-62795954
印 装 者:北京九州迅驰传媒文化有限公司
经　　销:全国新华书店
开　　本:185mm×260mm　**印　张**:25　**字　　数**:575 千字
版　　次:2014 年 9 月第 1 版　2016 年 9 月第 2 版　**印　　次**:2018 年 1 月第3 次印刷
印　　数:2601~4100
定　　价:49.00 元

产品编号:068926-01

出版说明

在教育部关于高等学校计算机基础教育三层次方案的指导下，我国高等学校的计算机基础教育事业蓬勃发展。经过多年的教学改革与实践，全国很多学校在计算机基础教育这一领域中积累了大量宝贵的经验，取得了许多可喜的成果。

随着科教兴国战略的实施以及社会信息化进程的加快，目前我国的高等教育事业正面临着新的发展机遇，但同时也必须面对新的挑战。这些都对高等学校的计算机基础教育提出了更高的要求。为了适应教学改革的需要，进一步推动我国高等学校计算机基础教育事业的发展，我们在全国各高等学校精心挖掘和遴选了一批经过教学实践检验的优秀的教学成果，编辑出版了这套教材。教材的选题范围涵盖了计算机基础教育的三个层次，包括面向各高校开设的计算机必修课、选修课以及与各类专业相结合的计算机课程。

为了保证出版质量，同时更好地适应教学需求，本套教材将采取开放的体系和滚动出版的方式（即成熟一本、出版一本，并保持不断更新），坚持宁缺毋滥的原则，力求反映我国高等学校计算机基础教育的最新成果，使本套教材无论在技术质量上还是文字质量上均成为真正的“精选”。

清华大学出版社一直致力于计算机教育用书的出版工作，在计算机基础教育领域出版了许多优秀的教材。本套教材的出版将进一步丰富和扩大我社在这一领域的选题范围、层次和深度，以适应高校计算机基础教育课程层次化、多样化的趋势，从而更好地满足各学校由于条件、师资和生源水平、专业领域等的差异而产生的不同需求。我们热切期望全国广大教师能够积极参与到本套丛书的编写工作中来，把自己的教学成果与全国的同行们分享；同时也欢迎广大读者对本套教材提出宝贵意见，以便我们改进工作，为读者提供更好的服务。

我们的电子邮件地址是 jiaoh@tup. tsinghua. edu. cn。联系人：焦虹。

清华大学出版社

第 2 版前言

作为在“中国大学 MOOC”平台开设的“大学计算机”MOOC 的配套教材及大面积实体课堂教学的主教材，本书第 1 版已经使用两年，作者有了一些体会，也发现了一些问题。本次修订，综合了由 MOOC 平台论坛上学习者的反馈建议和实际教学的体会，在原教材内容基础上，增加了有关数据结构方面的描述，并编写了配套的实验指导书。同时，继续保持了“基础＋问题求解”的整体架构，坚持自底向上的硬件系统构造思维培养和利用计算机求解问题能力的培养，即“计算思维”能力培养。

本书共包括 8 章。第 1 章首先带领读者走进计算机，了解计算机的组成和整体结构。然后从计算模型入手，讲述计算与可计算性基本理论、计算工具的发展以及基于计算机进行问题求解的一般过程。第 2 章从冯·诺依曼提出的二值符号体系出发，讲述计算机为什么采用二进制，以及不同信息在计算机中的表示与编码。试图从开关元件特性与 0 和 1 的对应引出逻辑的概念。第 3 章从基本逻辑运算及其门电路入手，借助推理和“搭积木”的思维模式，解析系统的“构造”过程。第 4 章从应用的角度讲述网络技术的一些基础知识，在网络无处不在的今天，了解这些是必要的。第 5 章和第 6 章是 C 语言编程技术。计算机唯一能够做的工作就是执行程序，要能够利用计算机解决各种问题，掌握一门程序设计语言、具备一定的编程能力是必要的。选择 C 语言作为学习程序设计的入门语言，主要考虑到它在算法描述上的优势，以及其有利于作为后续学习面向对象程序设计的基础。第 7 章是算法分析与设计，讲述算法的描述、算法复杂性评价及一些简单算法的设计方法，以帮助读者进一步理解第 1 章所述的可计算性理论，同时，通过亲自编程实现，使读者更深入地理解什么是算法，以及如何设计算法。第 8 章为数据结构基础，除介绍数据的逻辑结构、存储结构等基本概念外，主要讲述利用 C 语言实现线性表、栈和队列的设计方法。

总之，本书的宗旨是力求从计算、构造、设计的不同角度帮助读者初步建立和掌握利用计算机解决问题的思路和方法。

本书配有实验指导，其中除各项与主教材内容相关的基本程序设计和算法设计外，考虑到目前学生的实际情况，增加了部分主教材中未涉及的计算机基本应用技能的训练。

本书主要由吴宁(第 1～3 章)、崔舒宁(第 5～8 章)和夏秦(第 4 章)编写，吴宁负责统稿。本书在编写过程中得到首届国家级教学名师冯博琴教授的指点以及同事陈文革、杨振平、谢涛、贾应智等老师的帮助，作者在此表示衷心的感谢。

直至今天，大学本科新生的计算机基础水平依然存在较大差异，且这种差异会在可见的时间内长期存在。在分级教学难以实际操作的情况下，“大学计算机基础”这门课程教学内容的选取及相应教材的编写依然是难点。因此，由于这样的特殊性，加之作者水平所限，书中的不足和不妥之处在所难免，希望使用本教材的高校师生不吝指正。

作者

2016 年 6 月

第 1 版前言

1991 年,美国施乐公司 PARC 研究中心首席科学家 Mark Weiser 在 *Scientific American* 上发表了题为 *Computer for the 21th Century* 的文章,提出了“无处不在的计算(Ubiquitous Computing)”的理念,并由此开创了计算领域的第三次浪潮。无处不在的计算设备,无处不在的网络和通信,彻底改变了人类数千年的生活习惯。人们希望通过无处不在的计算,能随时随地获得自己希望的服务,且不用关心这些服务是怎样得到的。由于提供这些服务或计算的重要载体是计算机,因此,计算机成为人类生活中不可或缺的一部分。现代信息社会中的每一个人,无论从事何种工作,无论在学习什么专业,都需要学习使用计算机;而作为专业技术人员,更需要建立和掌握利用计算机求解各种专业问题的思路和方法,或者说,应具备计算思维的能力。

鉴于此,我们编写了这本以计算思维能力培养为出发点,围绕计算、构造和设计三大主题的“大学计算机”教材。本书与现有多数同类教材不同的是,除了不再追求“广而浅”的认知导向型模式,而转为具有针对性的“窄而深”的描述之外,首次从命题逻辑出发,讲述系统如何从基本逻辑门这样的“原子细胞”经过逐层封装与抽象,最终构成系统整体的思维过程。不仅帮助读者从构造的角度理解“抽象”“封装”这样一些软件理论中常见的概念,也在一定程度上培养这种自底向上的构造思维模式,这也是高等学校毕业生应具有的基本素质。

本书共 7 章。第 1 章首先带领读者走进计算机,了解计算机的组成和整体结构;然后从计算模型入手,讲述计算与可计算性基本理论、计算工具的发展以及基于计算机进行问题求解的一般过程。第 2 章从冯·诺依曼提出的二值符号体系出发,讲述了计算机为什么采用二进制,以及不同信息在计算机中的表示与编码。试图从开关元件特性与 0 和 1 的对应,引出逻辑的概念。第 3 章从基本逻辑运算及其门电路入手,借助推理和“搭积木”的思维模式,解析系统的“构造”过程。第 4 章从应用的角度讲述了网络技术的一些基础知识,在网络无处不在的今天,了解这些是必要的。第 5 章和第 6 章是 C 语言编程技术。实际上,计算机唯一能够做的工作就是执行程序,要利用计算机解决各种问题,掌握一门程序设计语言,具备一定的编程能力是必需的。选择 C 语言作为学习程序设计的入门语言,主要是考虑到它在算法描述上的优势,并利于作为后续学习面向对象程序设计的基础。第 7 章为算法分析与设计,讲述算法的描述、算法复杂性评价及一些简单算法的设计方法;希望能帮助读者进一步理解第 1 章所述的可计算性理论,同时,通过亲自编程实现,

能使读者更深入地理解什么是算法,以及如何设计算法。

总之,本书编写的宗旨就是力求从计算、构造、设计的不同角度,帮助读者初步建立和掌握利用计算机解决问题的思路和方法。

本书配备有实验指导书,实验指导书中除各项与主教材内容相关的基本程序设计和算法设计外,考虑到目前学生的实际情况,增加了部分主教材中未涉及的计算机基本应用技能的训练。

本书由吴宁(第1~3章)、崔舒宁(第5~7章)和陈文革(第4章)编写,吴宁负责统稿。本书在编写过程中得到首届国家级教学名师冯博琴教授的指点以及同事杨振平、谢涛、贾应智等老师的帮助,在此表示衷心的感谢。

虽然新生的计算机基础水平近年来已大有提高,但一个不争的事实是:直至今天,入校新生的计算机知识水平依然存在很大的差异,且这种差异会在可见的时间内长期存在。在分级教学难以实际操作的情况下,“大学计算机基础”这门课程教学内容的选取及相应教材的编写依然是难点。因此,由于这样的特殊性,但加之作者水平所限,书中错误和不妥之处在所难免,恳请广大师生不吝指正。

作者

2014年6月

目录

第1章 引论

引言

今天,计算机已成为人类生活不可缺少的一部分。现代信息社会中的每一个人都需要了解计算机,学习计算机科学。作为未来的工程技术人员和科学家,更需要具备利用计算机解决相关专业问题的能力,并能够从总体上判断什么样的问题是可以由计算机解决的。作为全书的引论,本章将首先带领读者走进计算机,“看看”什么是计算机;然后重点介绍计算机的理论模型,并由此入手,简要说明什么是计算以及可计算性理论的基本概念;关于计算工具的发展历史,有许多资料可以查阅,作为辅助的知识扩展,本章仅对此做一般性描述。帮助读者初步建立利用计算机求解问题的思路,掌握其基本方法,是本书编写的总体目标,这一思想贯穿于全书各章节。为便于后续内容的学习,本章先对基于计算机进行问题求解的基本过程进行总体描述。

教学目的

- 理解计算机系统的整体构成。
- 理解图灵机模型及其工作过程。
- 了解计算与可计算性。
- 了解计算工具的发展历程。
- 了解基于计算机的问题求解基本过程。
- 了解当前计算机科学研究的部分前沿技术。

1.1 走进计算机

计算机是20世纪人类最伟大的发明之一,在从诞生起至今的半个多世纪中,它由最初的“计算”工具,迅速发展成为应用于各行各业的信息处理设备,成为人类工作和生活中不可缺少的助手。现在,几乎每个年轻人都会使用计算机。但每位使用者是不是都了解计算机呢?

我们平时说的计算机,确切地说是计算机系统。因为没有人会认为不带显示器和键盘、鼠标的机器能称为"计算机"。其实,就计算机系统而言,它可以说是一个广义的概念。因为现代的计算机系统中,巨型机和微型机还是有比较多的差别,若融入网络技术和辅助的软件技术,如并行机、阵列机、机群系统,甚至今天比较"时髦"的云计算等,宏观上讲也都可以称为一个系统。

1.1.1 计算机系统构成

"计算机"是对一类系统的总称。它既可以指常见的个人计算机(Personal Computer,PC)或称微型计算机(如图 1-1 所示),也可以是计算速度达每秒几亿亿次的超级计算机(如图 1-2 所示)①。系统既包含可以看得见摸得着的硬件,也包含看得见却摸不着的各种软件。比如开机就可以看到的操作系统,或是文字编辑、游戏等各类应用软件。也就是说,计算机系统不仅包含物理上能够看得见的硬件实体,还包含运行于实体之上的、可实现各种操作功能的软件,即计算机系统是由硬件系统和软件系统两大部分组成的,其整体概念结构图如图 1-3 所示。由于人们经常直接接触和使用的计算机是微型计算机(microcomputer),所以,以下如无特殊说明,**本书中所说的计算机均特指微型计算机**,所介绍的计算机系统及基本结构和工作原理也均以微型计算机为蓝本。



图 1-1 个人通用台式计算机



图 1-2 天河二号超级计算机

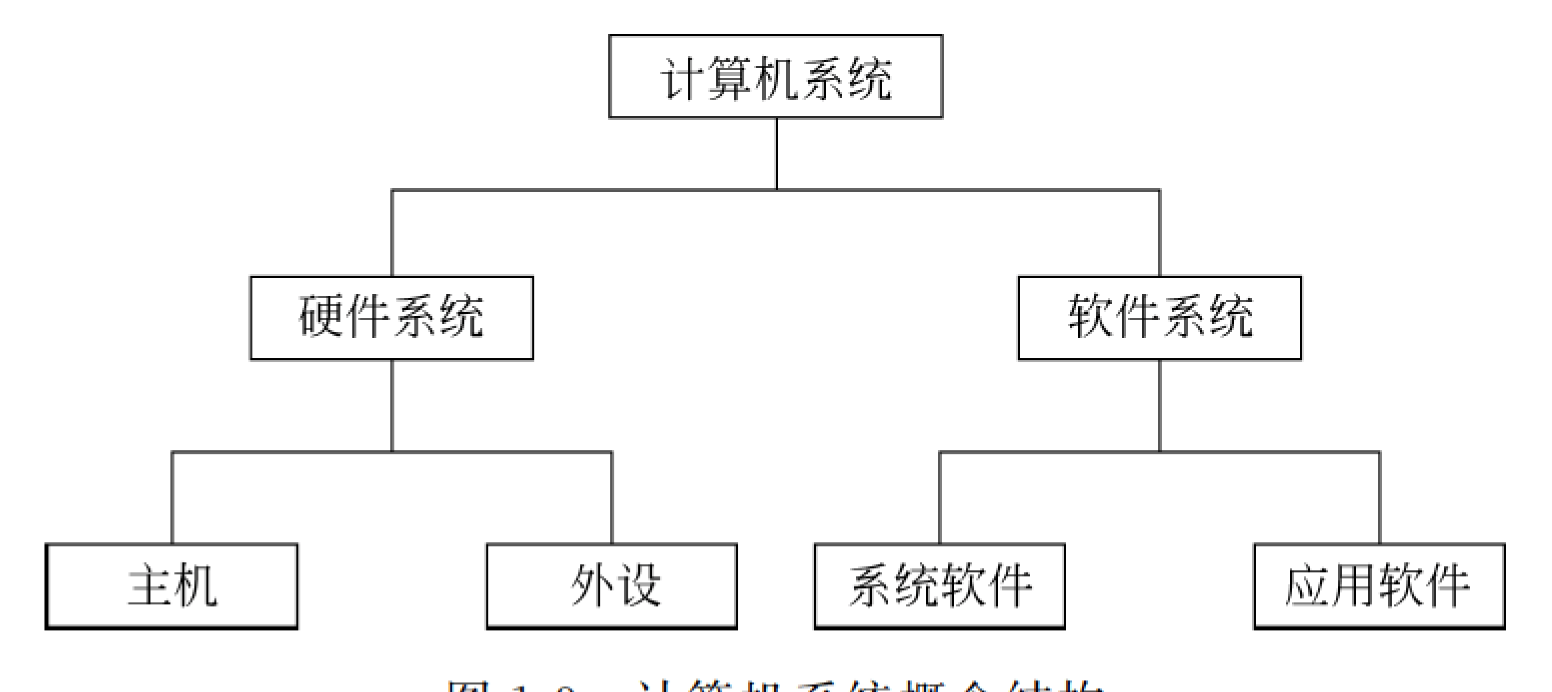


图 1-3 计算机系统概念结构

① 由中国国防科技大学研制的天河二号超级计算机,峰值计算速度为 5.49 亿亿次每秒,持续计算速度为 3.39 亿亿次双精度浮点运算每秒,在 2013 年 6 月德国莱比锡召开的国际超级计算机大会上,天河二号以其优异性能位居全球榜首。

1. 硬件系统

微机硬件系统包括主机和能够与主机进行信息交换的外部设备两部分。主机位于主机箱内，主要包括微处理器(CPU)、内存储器、I/O接口、总线和电源等。其中，微处理器是整个系统的核心。能否与处理器进行直接信息交换是主机部件的重要标志。所谓“直接信息交换”，就是不需通过任何中间环节(用专业术语说是接口)，就能够实现从处理器接收数据或向处理器发送数据。典型的如内存与处理器间的数据传输就是直接进行的。事实上，计算机正在运行的所有程序和数据，不论其曾经存放在哪里，在运行前都必须送入内存(也就是说，虽然文件都存放在硬盘上，但计算机操作这些文件时，它们是在内存中的，只是用户不知道它们是怎样“自动”进入内存的而已)，因为只有内存中的内容，处理器可以直接去“拿”(读取)、去“放”(写入)，从而保证计算机工作的高速度。

今天，如果有人告诉你他买了一台计算机，你一定会清楚他不是只抱了一台主机箱回来，至少还包括显示器、键盘和鼠标，这些都称为计算机的基本外部设备。

所谓外部设备，是指所有能够与计算机进行信息交换的设备(当然，这种信息交换需要通过接口)。它们既包括使用计算机所必需的基本外部设备(如上述键盘、鼠标、显示器等)，也包括其他各种能够连接到计算机、能够接收计算机所发送出的各种信息或向计算机发送信息的各类设备、控制仪器等。用于向计算机输入信息的称为输入设备，如键盘、鼠标器、扫描仪等；接收计算机输出信息的设备则称为输出设备，如显示器、打印机、绘图仪等。当然，也有些设备既能接收计算机输出的信息，也能向系统输入信息，如数码摄像机、硬盘(考虑一下：硬盘为什么是外部设备呢?)等，即它们兼具了输入设备和输出设备的功能，具体担当何种角色，则视其在某个时刻传送数据的方向。

相对于主机，外部设备的主要特点就是不能与处理器直接进行数据输入和输出，数据的传输必须通过接口进行。如硬磁盘，虽然安装在主机箱内，但不属于主机系统，因为它与处理器的通信需要通过专用接口进行。而至于什么是接口，会在1.1.2节中给出简单的介绍。

有关计算机常用外设的基本工作原理参见附录A。

2. 软件系统

硬件系统是计算机工作的物理基础，但要使其正常工作并完成各种任务，还必须有相应的软件支撑。所谓软件，不仅仅是一般概念中的程序，而是程序、数据以及相关文档的总称。这里，数据是程序处理的对象，文档是指与程序开发、维护和使用有关的各种图文资料。软件可以分为两大类：系统软件和应用软件。

系统软件是管理、监控和维护计算机软硬件资源的软件，由计算机设计者提供，包括操作系统和各种系统应用程序。操作系统(Operating System，OS)是配置在计算机硬件上的第一层软件，是其他软件运行的基础。其主要功能是管理计算机系统中的各种硬件和软件资源(如存储器管理、文件管理、进程管理、设备管理等)，并为用户提供与计算机硬件系统之间的接口(如通过键盘发出命令控制作业运行等)。在计算机上运行的其他所有的系统软件(如编译程序、数据库管理系统、网络管理系统等)及各种应用程序都要依赖于

操作系统的支持。因此,操作系统是计算机中必须配置的软件,在计算机系统中占据着极其重要的位置。目前较为流行的操作系统有 Windows 系列、UNIX、Linux 以及苹果计算机采用的 Mac 系列等。

系统应用程序运行于操作系统之上,是为应用程序的开发和运行提供支持的软件平台。主要包括以下几类:

(1) 编译程序。用于将用各种计算机语言(汇编语言或各种高级语言)编写的程序翻译成计算机硬件能够直接识别的用二进制码表示的机器语言。由于计算机硬件是由各种逻辑器件构成的,只能识别电脉冲信号,也就是 0 和 1 组成的二进制码,这种由二进制码组成的计算机语言称为机器语言,人类很难理解和记忆。目前广泛使用的计算机程序设计语言都是接近人类自然语言的高级语言,为了使计算机能够理解,必须要经过一个翻译的过程,而编译程序的功能就是实现这样的翻译。

(2) 计算机的监控管理程序(monitor)、故障检测和诊断程序,以及调试程序(debug)。它们负责监控和管理计算机资源,并为应用程序提供必要的调试环境。

(3) 各类支撑软件,如数据库管理系统及各种工具软件等。

应用软件是应用程序员利用各种程序设计语言编写的、面向各行各业实现不同功能的应用软件,如工程设计程序、数据处理程序、自动控制程序、企业管理程序等。目前,软件的设计还没有摆脱手工操作的模式,但随着软件技术的进步,应用软件也在逐渐地向标准化、模块化方向发展,目前已形成了部分用于解决某些典型问题的应用程序组合,称为软件包(package)。

软件系统的核心是系统软件,系统软件的核心则是操作系统。

计算机系统是硬件和软件的结合体,硬件和软件相辅相成,缺一不可。硬件是计算机工作的物质基础,而软件是计算机的灵魂。没有硬件,软件就失去了运行的基础和指挥对象;而没有软件,计算机就不能工作,其效能就不能充分发挥出来。

对某项具体任务,通常既可以用硬件完成,也能通过软件完成。从理论上讲,任何软件算法都能用硬件实现,反之亦然,这就是软件与硬件的逻辑等价性。设计计算机系统或是在现有的计算机系统上增加功能时,具体采用硬件还是软件实现,取决于价格、速度、可靠性等因素。早期的计算机受技术和成本的限制,硬件都相对简单。如今,随着超大规模集成电路技术的发展,以前由软件实现的功能现在更多地直接用硬件实现,为的是提高系统的运行速度和效率。另外,在软件和硬件之间还出现了所谓的固件(firmware),它们在形式上类似硬件,但在功能上又像软件,可以编程和修改,这种趋势称为软件的硬化和固化。

1.1.2 主机与主机板

1. 主机

PC 的硬件系统除了各种类型的外部设备之外,最主要的就是主机系统了,它是组成微型计算机的主体。主机系统的主要部件包括中央处理单元(Central Processing Unit,

CPU)、内存储器、总线和输入输出接口。其基本组成如图 1-4 所示。

1) CPU

中央处理单元(CPU)在微机中也称为微处理器(microprocessor),是微型计算机的核心芯片,也是整个系统的运算和指挥控制中心。它的基本功能就是按照程序一条一条地执行程序,每一条指令规定一个基本的操作,并且与相应的物理电路对应,这就是计算机硬件和软件之间最基本的连接,也是计算机软、硬件协同的基础。有关什么是指令以及 CPU 的内部结构,将在 3.2 节中做具体的介绍。

CPU 是人类借助超大规模集成电路技术生产的最复杂、最精细的产品之一。CPU 的种类很多,除了微型计算机中的微处理器之外,各种网络服务器、巨型机等设备中的高性能处理器,则是计算能力更强的 CPU。另外,安装在各种现代化仪器设备和通信设备内的处理器则称为嵌入式 CPU,目前几乎所有的高档电器内部都装备了一片或几片这种嵌入式 CPU。

无论哪一种 CPU,其内部总体上都主要包括三大部分,即运算器、控制逻辑单元(或称控制器)和内部寄存器组(如图 1-5 所示),各部分通过 CPU 内部总线连接在一起,现代微处理器中还包含高速缓冲存储器(cache)。

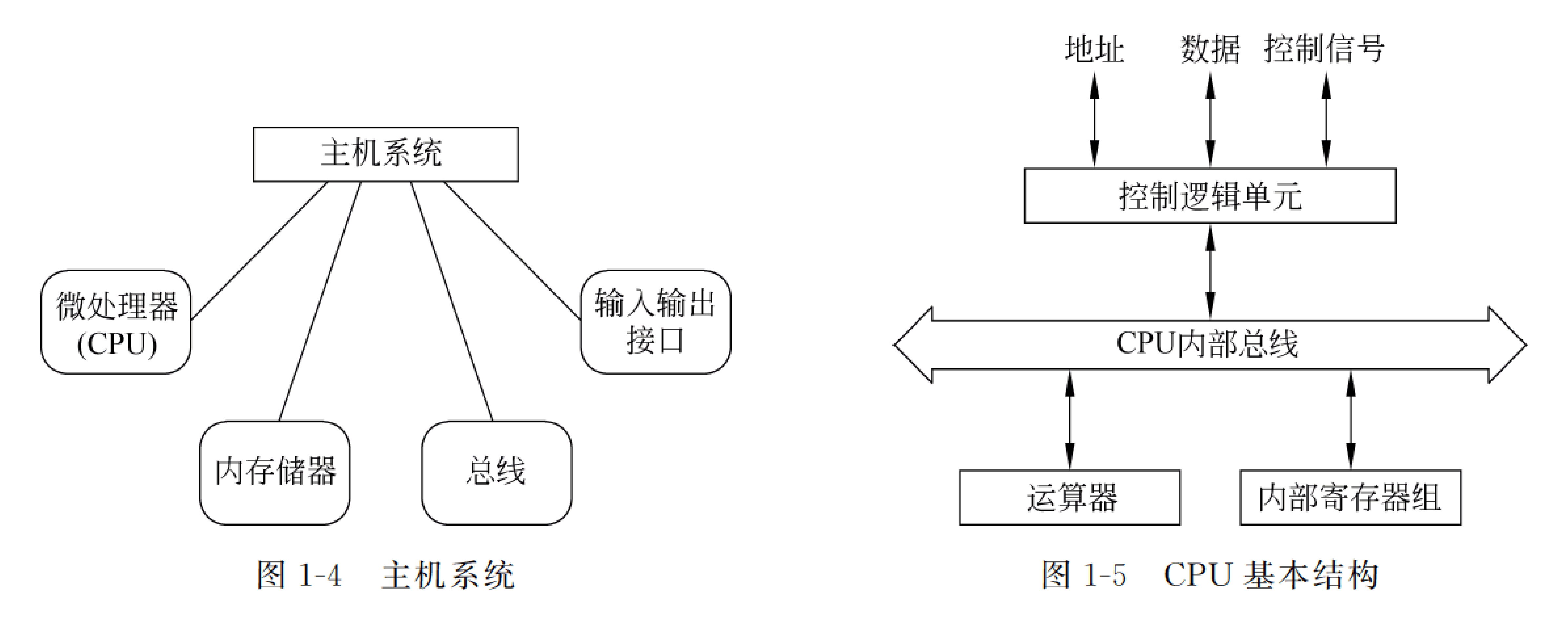


图 1-4　主机系统　　　图 1-5　CPU 基本结构

运算器的主要部件是算术逻辑单元(Arithmetic Logical Unit,ALU),它是运算器的主体。ALU 的主要功能就是在控制信号的作用下可完成加、减、乘、除等算术运算、移位操作及各种逻辑运算(扩大一点讲就是执行指令),现代新型 CPU 的运算器还可完成各种浮点运算。运算产生的中间结果可以存放在 CPU 的内部寄存器中。

内部寄存器组是 CPU 内部的若干个用于暂时存放数据的存储单元。包括多个通用寄存器和若干专用寄存器。寄存器的功能按其字面意思可理解为用来暂时存放数据的部件(当然,这里的“数据”是广义的,并非仅仅是数值),“暂存数据”这一任务主要由通用寄存器完成。设置它们的目的是在执行程序过程中,对某些需要重复使用的操作数据或中间运算结果,可将它们暂时存放在寄存器中,以避免对存储器的频繁访问,从而缩短执行时间。专用寄存器的作用则是固定的。

为了便于本书后续内容的学习,有一个专用寄存器需要在这里先做初步的介绍,这就是程序计数器(Program Counter,PC)。程序计数器用于指示下一条要取指令的地址。不论是程序员编写完成的程序,还是人们日常编写的各种文档,通常都是存放在外存储器

(如硬盘)中,它们在被CPU执行之前,都需要先进入内存[①]。而CPU在执行的时候,为什么就能够按照人们要求的顺序去执行程序、处理文档呢?靠的就是程序计数器(PC)的控制。所以,它是非常重要的一个内部寄存器[②]。

控制逻辑单元主要用于控制和协调整个CPU的工作,是整个CPU的指挥控制中心。人们日常的上下班是以时间为基准的,CPU的工作基准是时钟信号。这是一组周期恒定的脉冲信号。不同的时刻CPU做不同的工作,它们在时间上有着严格的关系,这就是时序。时序信号由控制器产生,控制CPU的各个部件按照一定的时间关系有条不紊地完成指令要求的操作。

各种数据在CPU中的传送要借助于CPU的内部通道,这个通道称为总线(bus)。

从20世纪70年代第一片处理器诞生至今,CPU的性能不断提高。近年来,人们又开发了一种具有新型体系结构的CPU——多核CPU。所谓多核,指的是在一个芯片上集成多个物理的CPU运算内核(即运算器),这些运算内核可以并行、协同地工作(这就相当于将一件复杂的工作分给多个人同时工作一样),从而使计算机的处理能力大大增强。多核CPU和以前的单核CPU在外观上并没有太大的区别,但其内部结构已是大不相同了。图1-6是Intel Pentium 4单核处理器Intel Core i7四核处理器。

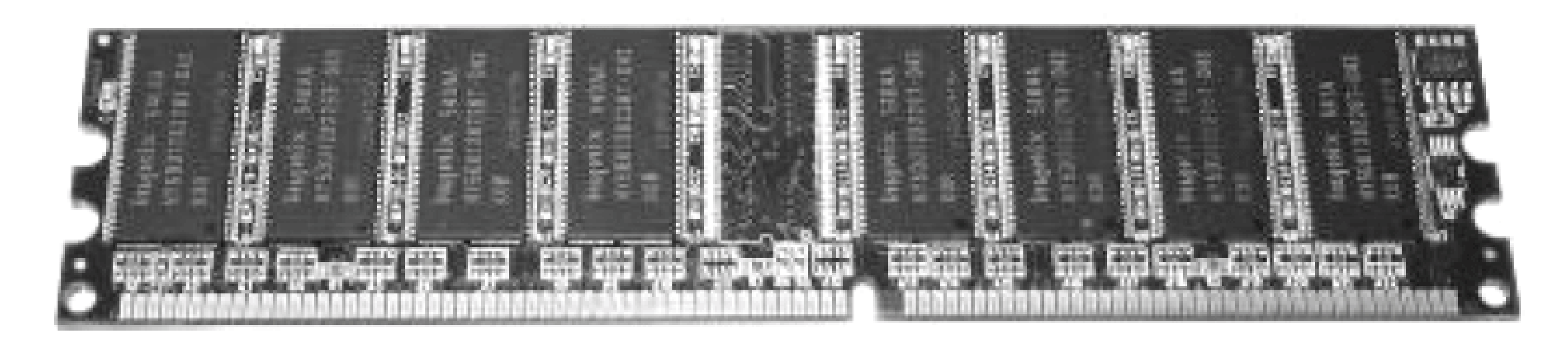


图1-6 微处理器外观

2)存储器

存储器(memory)是计算机中用于存储各种信息的部件,总体上可以分为内存和外存两种类型。对存储器的操作有两种,即“读”和“写”。“读”表示从存储器中输出数据,也称为读取;“写”表示向存储器输入数据,也称为写入。对存储器的读写可以按字节、字或块进行。

内存储器由半导体材料制成(所以也称半导体存储器),属于主机部分,用于存放计算机当前运行的程序(包括确保计算机运行所必需的程序)及运算的数据。我们常说的内存条(如图1-7所示)就是主要的内存储器,称为主存。


图1-7 内存条

相对于外存储器,内存的主要特点有:

---

① 有关编写的程序如何进入内存以及为什么要进入内存才能被执行,请参阅操作系统方面的书中有关进程管理部分的描述。

② 要真正理解程序计数器的作用,需要深入理解微型计算机的工作原理。作为计算机基础教材,本书仅在第3章对微型机的基本工作原理做一点初步的介绍,若希望深入理解,需要进一步学习其他书籍。

(1) 可以与 CPU 直接进行信息交换(所以,它属于主机部分)。

(2) 存取速度快,容量小,单位字节容量价格较高。

(3) 需要后备电源,当断电时,其上存放的信息将丢失。

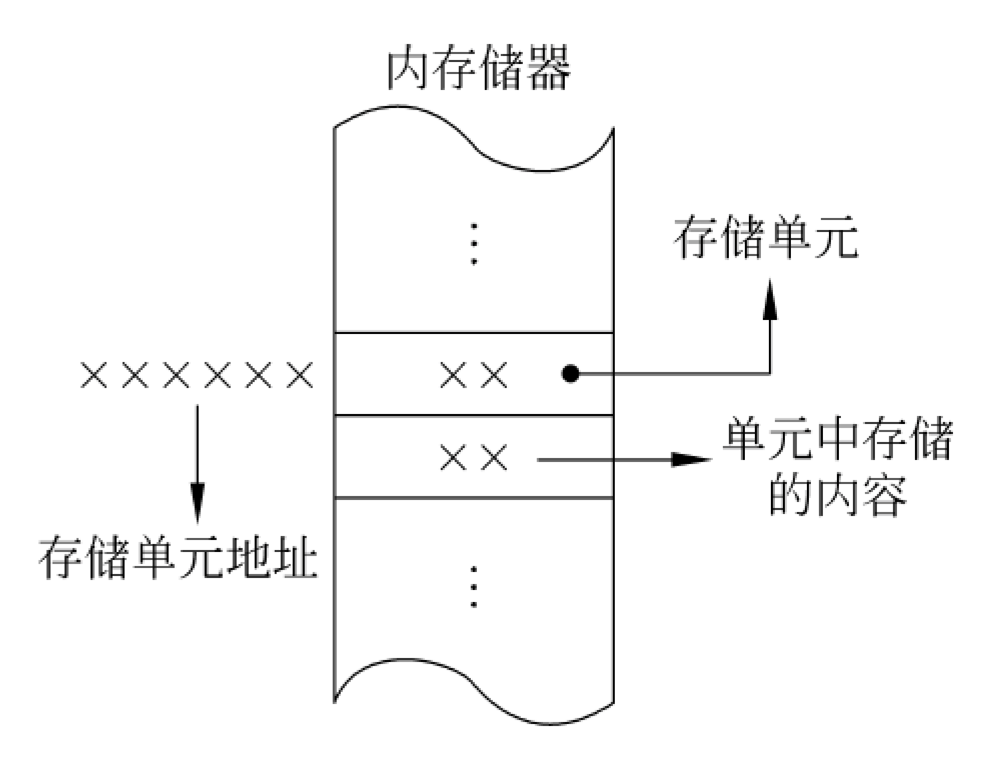


图 1-8 内存结构示意图

如同一栋大楼是由若干个房间组成一样,内存由若干个单元组成(如图 1-8 所示)。大楼中的每个房间都有门牌号码,且每个号码在楼内都是唯一的,目的是便于寻找;同样,内存中的每个单元也有“门牌号码”,称为地址码,每个单元的地址在内存中也是唯一的。由于计算机只能识别二进制,所以内存中的地址码都是用二进制表示。地址码的长度依内存单元的个数(称为容量)而定。比如,4 个单元的内存的地址码只需要 2 位二进制就可以表示(明白为什么吗),而 4G 个单元的内存中,每个单元的地址则需要 32 位二进制码来表示。

在微机系统中,内存的每个单元都存放 8 位二进制码,即 1B 数据。内存的容量就是指它具有的单元数。如常说的 2GB 内存,意思就是该内存有 2G(1G=2^{30})个单元,每个单元中有 1 字节数据。

对内存的读/写操作通常按“字”进行,不同的系统“字”的长度不同。目前的微型机多为 64 位机,其在一个周期中能够对内存读出或写入 8B 数据。

外存储器包括联机外存和脱机外存两种,用于存放 CPU 不直接执行,但可以长期保留的数据。脱机外存有光驱、磁带、移动存储器等,由复合材料(如光盘)、磁性材料或半导体材料(如优盘)构成。它们可以脱离计算机而存在,所以理论上可以存放无限多的数据。联机外存就是人们常说的硬盘。

硬盘是微机中主要且必备的存储部件,由多片磁性材料制造的盘片叠加在一起构成(如图 1-9 所示)。每个盘片有两个记录面(每个记录面对应一个磁头,所以也用磁头数表示记录面数),每个记录面上是一系列称为磁道的同心圆(多个记录面上的同心圆叠放在一起就构成柱面),每个磁道又被划分为若干个扇区(sector)。硬磁盘就是按记录面、磁



(a) 硬盘外观



(b) 硬盘内部结构

(c) 硬盘原理示意

图 1-9 硬磁盘及其及内部结构

道和扇区来对数据进行组织的。对硬磁盘进行读/写操作的基本单位是扇区，每个扇区的容量为512B，而整块硬盘的容量则为

$$磁头数\times柱面数\times扇区数\times512(B) \tag{1.1}$$

【例 1-1】 设已知磁头数为16，柱面数为4096，扇区数为64，求该硬盘的容量。

由式(1.1)可得：

$$该硬盘的容量=16\times4096\times64\times512=2\ 147\ 483\ 648B=2GB$$

外存储器的主要作用是保存各种希望由计算机保存和处理的信息。相对于内存，外存具有存储容量大(目前微机硬磁盘的常规配置为1TB，而内存通常为4GB或8GB)、速度慢、单位字节容量价格低、不能与处理器直接进行信息交换等特点。外存储器虽然也安装在主机箱中，但属于外部设备的范畴。理由是外存与处理器之间的信息交换需要通过输入输出接口。目前微型机中最常用的硬盘接口标准是SATA(Serial Advanced Technology Attachment)，它定义了外存储器(如硬盘、光盘等)与主机的物理接口。

3) 输入输出接口

上面已提到，凡与处理器之间的信息交换需要通过输入输出接口的设备都称为外部设备(比如硬磁盘)或输入输出设备。我们很容易想到，不能连接外部设备的计算机是没有意义的(你能够想象没有显示器、鼠标和键盘的计算机吗)。

所有能够与计算机通信的设备都可以称为输入输出设备，它们虽然千差万别，种类繁多，结构和原理各异，但都有一个共同的特点，就是：要想接收来自CPU的信息或将信息送入CPU去处理，必须通过一个中间环节，就是输入输出接口(Input/Output Interface，简称I/O接口)，也称I/O适配器(I/O adapter)。

I/O接口是将外设连接到系统总线上的一组逻辑电路的总称，也称为外设接口。其在系统中的作用如图1-10所示。在一个实际的计算机控制系统中，CPU与外部设备之间常需要进行频繁的信息交换，包括数据的输入输出、外部设备状态信息的读取及控制命令的传送等，这些都是通过接口来实现的。由I/O接口在系统中的位置，使得接口电路应解决如下问题，这也是接口应具有的功能：

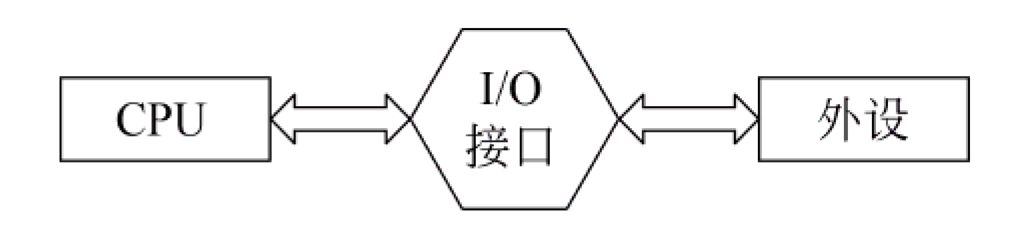


图1-10 I/O接口在系统中的作用示意图

(1) CPU与外设的速度匹配。CPU与外设之间的工作时序和速度差异很大，要使两者之间能够正确进行数据传送，需要接口做“适配”。接口电路应具有信息缓冲能力，不仅应缓存CPU送给外设的信息，也要缓存外设送给CPU的信息，以实现CPU与外设之间信息交换的同步。

(2) 信息的输入输出。通过I/O接口，CPU可以从外部设备输入各种信息，也可将处理结果输出到外设。同时，为保证数据传输的正确性，需要有一定的监测、管理、驱动等能力。

(3) 信息的转换。外部设备种类繁多，其信号类型、电平形式等与CPU都可能存在差异。I/O接口应具有信息格式变换、电平转换、码制转换、传送管理以及联络控制等功能。

(4) 总线隔离。为防止干扰，I/O接口还应具备一定的信号隔离作用，使各种干扰信

号不影响 CPU 的工作。

2. 主机板

主机板也称主板(mainboard)或系统板(system board),是微型机的物理构成,在整个微机系统中扮演着举足轻重的角色。可以说,主板的类型和档次决定着整个微机系统的类型和档次,主板的性能影响着整个微机系统的性能。

主机板在结构上主要有 AT 主板、ATX 主板、NLX 主板和 BTX 主板等类型。它们之间的区别主要在于各部件在主板上的位置排列、电源的接口外形、控制方式及尺寸等。不论哪种结构,均采用开放式结构。可以通过更换安装在扩展插槽上的外围设备控制卡(适配器),实现对微机相应子系统的局部升级。图 1-11 为一个实际的 ATX 主板的布局结构及外形图。

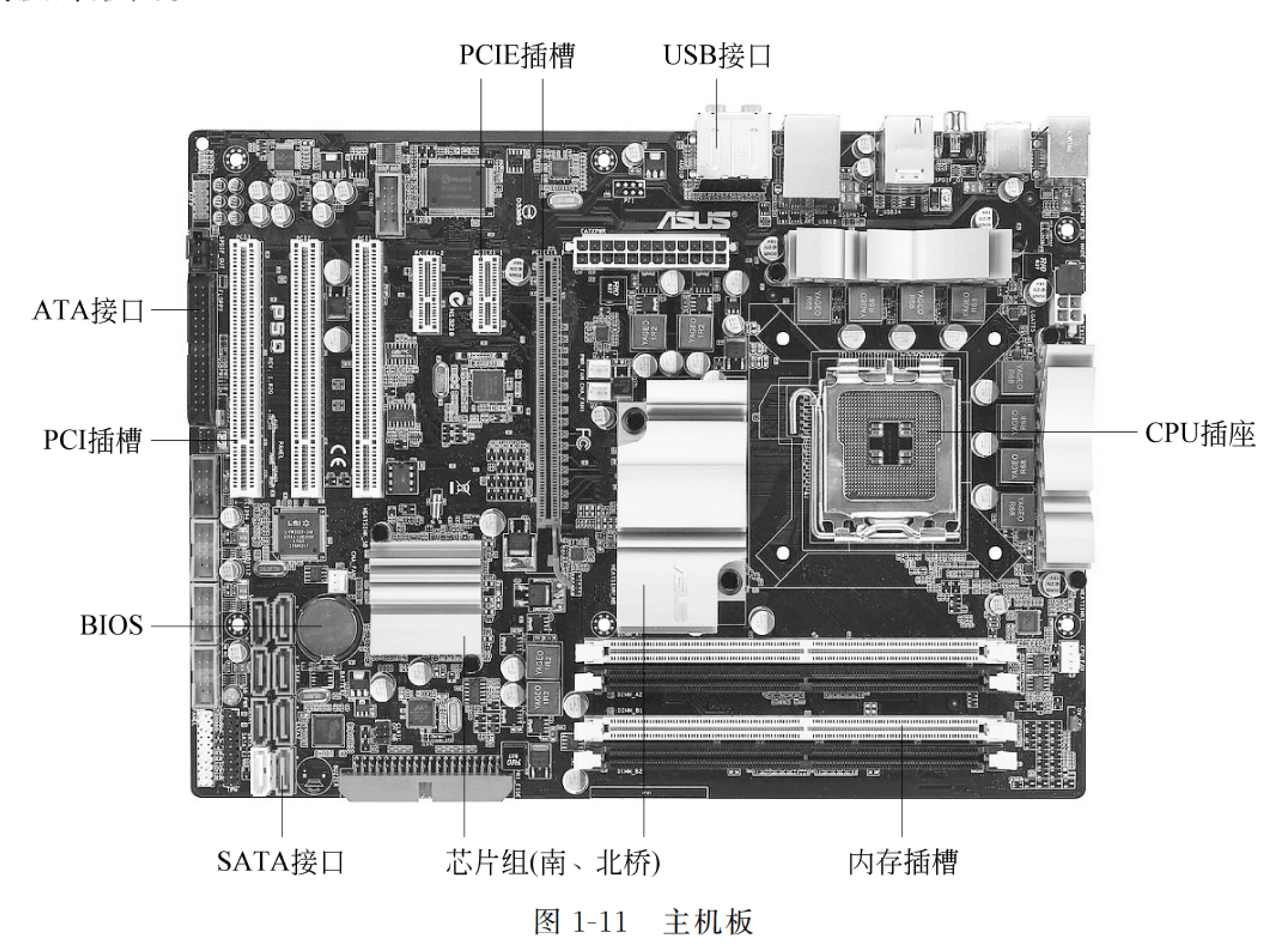


图 1-11　主机板

主板位于主机箱内,上面安装了组成计算机的主要电路系统,主要包括芯片、扩展槽和对外接口 3 种类型的部件。

1) 芯片部分

芯片部分除微处理器(CPU)外,主要有控制芯片组和 BIOS。

芯片组是主板上一组超大规模集成电路芯片的总称,是主板的关键部件,用于控制和协调计算机系统各部件的运行,它在很大程度上决定了主板的功能和性能。可以说,系统的芯片组一旦确定,整个系统的定型和选件变化范围也就随之确定。

典型的芯片组由北桥芯片和南桥芯片两部分(2 片芯片)组成,故也称南北桥芯片。图 1-11 中 CPU 插槽旁边被散热片盖住的就是北桥芯片。北桥芯片是芯片组的核心,主要负责处理 CPU、内存、显卡三者间的"交通",由于其发热量较大,故需加装散热片。南

桥芯片主要负责硬盘等存储设备和PCI之间的数据流通。

BIOS也称系统BIOS，是一块方块状的存储器芯片，里面存有与该主板搭配的基本输入输出系统程序，能够让主板识别各种硬件，还可以设置引导系统的设备，调整CPU外频等。BIOS芯片是可读写的只读存储器（EPROM或E^2PROM）。机器关机后，其上存储的信息不会丢失。在需要更新BIOS版本时，还可方便地写入。当然，BIOS不利的一面便是会让主板遭受病毒的袭击。

系统BIOS程序主要包含以下几项功能：

（1）上电自检（Power-On Self Test，POST）。在微机加电后，CPU从地址为0xFFFFFF0H处读取和执行指令，进入加电自检程序，测试整个微机系统是否工作正常。

（2）初始化。包括可编程接口芯片的初始化；设置中断向量表（一个专门用于存放中断程序入口地址的内存区域）；设置BIOS中包含的中断服务程序的中断向量（即将这些中断程序入口地址放入中断向量表中）；通过BIOS中的自举程序将操作系统中的初始引导程序装入内存，从而启动操作系统。

（3）系统设置（setup）。装入或更新CMOS RAM保存的信息。在系统加电后尚未进入操作系统时，按Del键（或其他热键）可进入Setup程序，修改各种配置参数或选择默认参数。

2）扩展槽

安装在扩展槽上的部件属于可插拔部件。所谓“可插拔”是指这类部件可以用“插”来安装，用“拔”来卸除。主板上的扩展槽包括内存插槽和总线接口插槽两大类。

内存插槽一般位于CPU插座下方，用于安装内存储器（也就是内存条）。通过在内存插槽上插入不同的内存条，就可方便地构成所需容量的内存储器。主板上内存插槽的数量和类型对系统主存的扩展能力及扩展方式有一定影响。

总线接口插槽是插接各种扩展接口卡的地方，是CPU通过系统总线与外部设备联系的通道，主要有PCI插槽、AGP插槽或PCI Express（PCIE）插槽。

所谓**总线**，可以简单地说是计算机中传输信息的通道。微机中的总线按照其层次结构，可以分为CPU总线（或称前端总线）、系统总线和外设总线。

前端总线一般是指从CPU引脚上引出的连接线，用来实现CPU与主存储器、CPU与I/O接口芯片、CPU与控制芯片组等芯片之间的信息传输，也用于系统中多个CPU之间的连接。前端总线是生产厂家针对其具体的处理器设计的，与具体的处理器有直接的关系，没有统一的标准。

系统总线也称为I/O通道总线，是主机系统与外围设备之间的通信通道。在主板上，系统总线表现为与总线接口插槽（也称I/O插槽）的引线连接的一组逻辑电路和导线。I/O插槽上可插入各种扩展板卡，它们作为各种外部设备的适配器与外设相连。为使各种接口卡能够在各种系统中实现即插即用，系统总线的设计要求与具体的CPU型号无关，而有自己统一的标准，各种外设适配卡可以按照这些标准进行设计。目前常见的总线标准有PCI总线、PCI-E总线等。

PCI 插槽多为乳白色，是主板的必备插槽，可以插入声卡、网卡、多功能卡等设备。AGP 插槽的颜色多为深棕色，位于北桥芯片和 PCI 插槽之间，用于插入 AGP 显卡，有 1X、2X、4X 和 8X① 之分。在 PCI Express 出现之前，AGP 显卡是主流显卡，其传输速度最高可达到 2133MB/s(AGP8X)。随着 3D 性能要求的不断提高，AGP 总线的传输速度已越来越不能满足视频数据处理的要求。在目前的主流主板上，显卡接口多选择 PCI Express。PCI Express 插槽有 1X、2X、4X、8X 和 16X 之分。

外设总线是指计算机主机与外部设备接口的总线，实际上是一种外设接口标准。目前在微机系统中最常用的外设接口标准就是 USB(Universal Serial Bus，通用串行总线)，可以用来连接多种外部设备。

图 1-12 为现代微型机中的总线结构示意图。

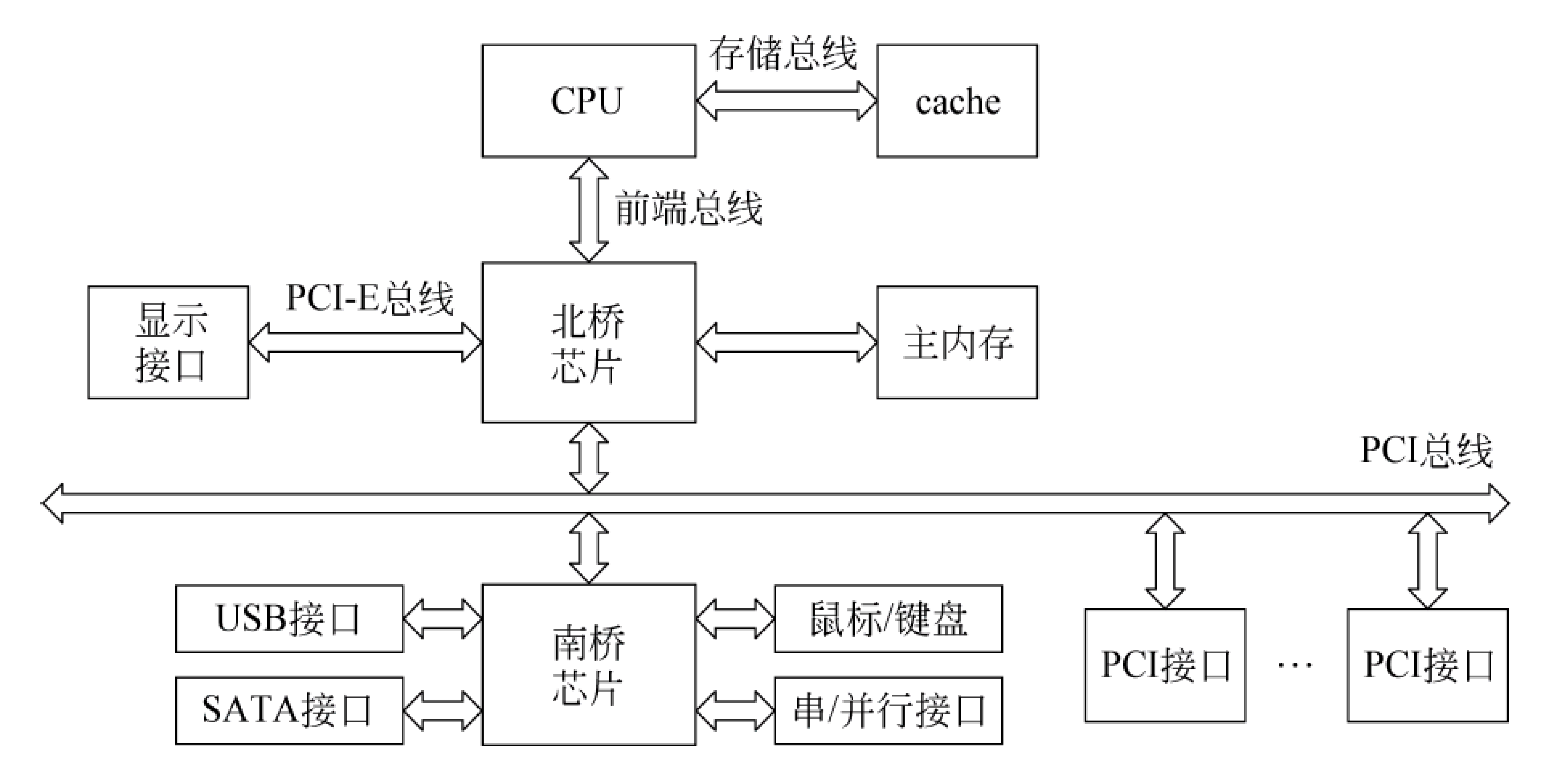


图 1-12 现代微型机中的总线结构

除上述这些主要部件外，主板上还有用于连接硬盘、光驱等的电缆插座、键盘/鼠标接口以及许多不可缺少的逻辑部件和跳线开关等。所有这些部件密切联系、相互沟通，实现了整个微型机中各部件间的数据交流。

1.1.3 计算机的主要性能指标

衡量一个微处理器性能的高低，最重要的是执行指令(或程序)所用时间的多少。而所用时间的多少又与时钟速度和执行一条指令所需的时钟脉冲个数有关。微处理器的时钟速度越快，执行指令需要的时钟脉冲个数越少，指令执行的速度就越快。这也是为什么在同等情况下，CPU 的钟频越高，运算速度越快的原因。

表征微机系统性能的指标较多，这里简要介绍其中的几项。

1. 主频

主频是主时钟频率的简称，指在一秒钟内发生的同步脉冲数，单位为兆赫(MHz)。

① nX 表示 n 倍速，即对原来的时钟脉冲进行技术处理后，使时钟频率变成 n 倍频。

主频很大程度上决定了计算机的运行速度，主频越高意味着计算机的速度也越快。

2. 运算速度

程序由一条条指令组成（详见 3.3 节），执行一条指令所花费的时间越少，计算机的工作速度就越高。衡量计算机针对整数的运算速度用 MIPS（Million Instructions Per Second，每秒百万条指令）表示；对于浮点运算，一般使用 MFLOPS（Million FLoating point Operations Per Second）表示，即每秒百万次浮点运算。

3. 内存容量

内存容量指内存存储数据的能力。存储容量越大，CPU 能直接访问到的数据就越多。存储器最基本的计量单位是字节（B），一个字节由一个 8 位（8b）二进制数组成，此外还有 KB、MB、GB 和 TB 等存储容量单位。

4. 字长

字长指 CPU 能够同时处理的二进制位数。字节越长，运算精度越高，数据处理速度越快。

5. 外部设备的配置及扩展能力

外部设备的配置及扩展能力主要指计算机系统连接各种外部设备的可能性、灵活性和适应性。常见配置有 C 盘驱动器的配置、硬盘接口类型与容量、显示器的分辨率等。

1.2 图灵机模型与计算问题

1.2.1 图灵机模型

今天，计算机已深入到生活和工作的各个领域，在享受着计算机所带来的诸多便利的同时，需要记住两位对计算机科学的发展做出了巨大贡献的人，一位是计算机理论的奠基人艾伦·麦席森·图灵（Alan Mathison Turing）（见图 1-13），另一位则是现代计算机体系结构的设计者冯·诺依曼（John von Neumann）。

图灵是英国著名的数学家和逻辑学家，他一生所做出的最大的贡献就是设计了理论计算机，第一次将“算法”（第 7 章会详细介绍）这样一个又基本、又深刻、在当时已被讨论了近 30 年但没有明确定义的概念用一个模型讲清楚了。这一点成为后人设计实用计算机的思路来源和理论基石，因此，图灵也被称为计算机理论之父。

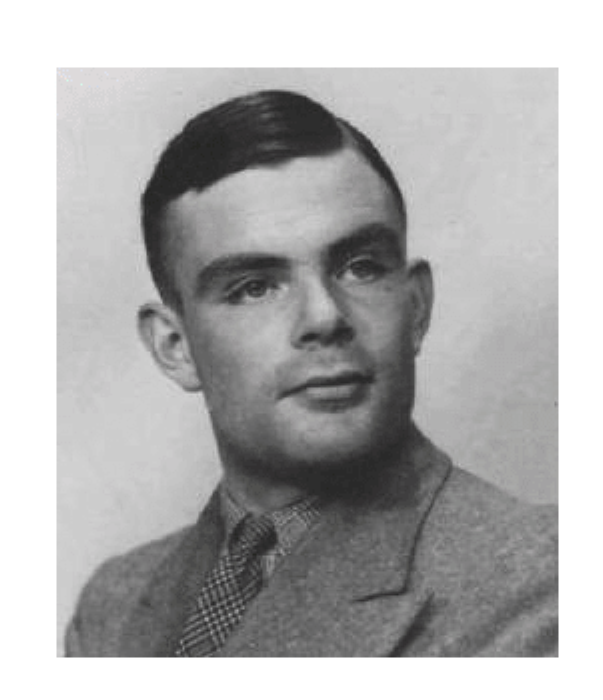


图 1-13　图灵

1936 年，图灵发表了一篇题为《论可计算数及其在判定问题中的应用》（*On Computable Numbers, with An Application*

to the Entscheidungsproblem)的论文,从一个全新的角度给出了可计算函数[①]的定义,也就是说明了什么问题是可计算的。他全面分析了人的计算过程,将计算归结为最简单、最基本、最确定的操作动作,从而用一种简单的方法来描述那种直观上具有机械性的基本计算程序,使任何机械(能行)的程序都可以归约为这些动作。这种简单的方法是以一个抽象自动机概念为基础的,其结果是:算法可计算函数就是这种自动机能计算的函数。这不仅给计算下了一个完全确定的定义,而且第一次把计算和自动机联系起来,对后世产生了巨大的影响,这种自动机后来被人们称为图灵机。

1. 图灵机

图灵的理论计算机也称为图灵机(Turing Machine,TM)。图灵机不是一种具体的机器,而是一种思想模型,所以也称为图灵模型。它的基本思想就是:用机器来模拟人们用笔和纸进行数学运算的过程。或者说,图灵机是将计算与自动进行的机械操作联系在一起的一种模型。

图灵将人的计算过程看作两个简单的动作:

① 在纸上写上或擦除某个符号;

② 将注意力从纸上的一个位置移动到另一个位置,而人每一次的下一步动作走向依赖于人当前所关注的纸上某个位置的符号及人当前的思维状态。

为了模拟人的这种运算过程,图灵构造出一台假想的(抽象的)机器(如图 1-14 所示),该机器由以下几个部分组成:

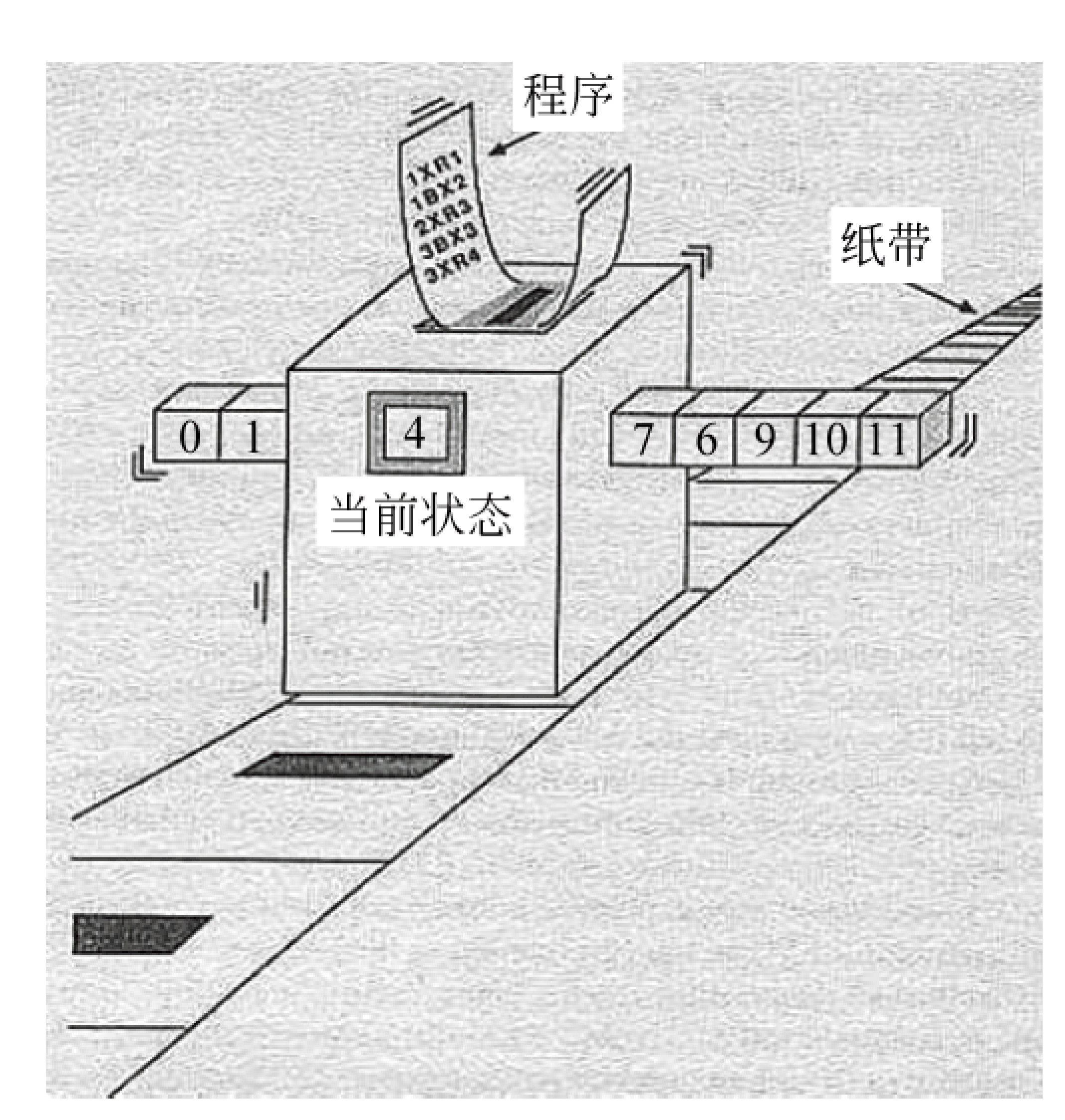


图 1-14　图灵机结构模型

(1) 一条右端可无限延长的纸带(type)。纸带被划分成一个个连续的方格,称为单元格(cell)。每个单元格中可包含一个来自有限字母表的符号(称为带符,tape symbol),字母表中有一个特殊的符号表示空白。

(2) 一个读写头(Head,图 1-14 中间的大盒子)。读写头内部包含了一组固定的状态(盒子上的方块)和程序。该读写头可以在纸带上左右移动,它能读出当前所指的格子上的符号,并能改变当前格子上的符号。

(3) 一套控制规则(Table,即程序)。规则包括当前读写头的内部状态、输入数值、输出数值、下一时刻的内部状态。在每个时刻,读写头都从当前纸带上读入一个方格信息。根据当前机器所处的状态及读写头所读入的格子上的符号来确定读写头下一步的动作。同时,改变状态寄存器的值,令机器进入一个新的状态。

(4) 一组内部状态(图 1-14 中穿过大盒子的小方块)。它用来保存图灵机当前所处

① 依据丘奇-图灵论题,可计算函数精确的定义是:使用可给出无限数量时间和存储空间的计算设备来计算的函数。或等价地说,有算法的任何函数都是可计算的。

的状态。图灵机的所有可能状态的数目是有限的，并且有一个特殊的状态，称为停机状态。

2. 图灵机的工作过程

将图灵机模型画成二维平面图如图 1-15 所示。图中，X_i 表示单元格中的带符，也是输入符号(就是可以被读写头读入的符号)。B 表示空格，它是带符，但不是输入符号。若读写头位于某个单元格之上，说明图灵机正在读写这个单元。

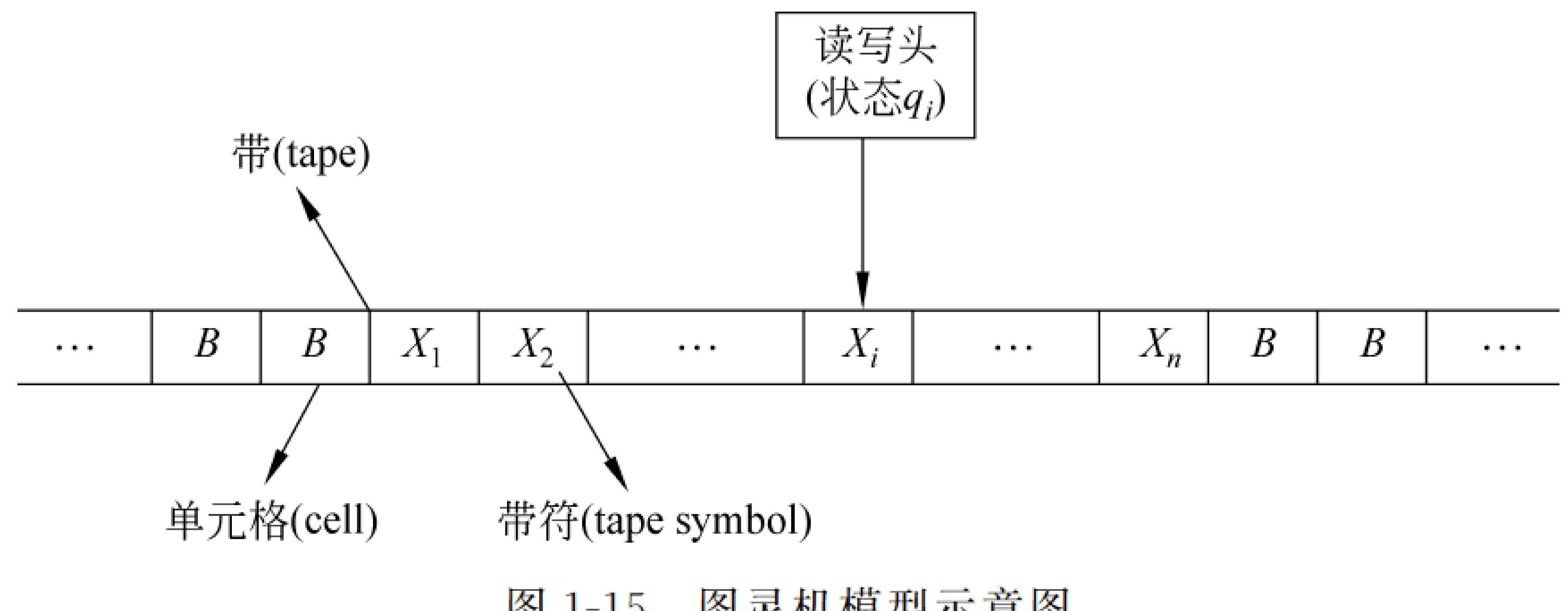


图 1-15　图灵机模型示意图

下面用一个简单的示例来说明图灵机的工作过程。

【例 1-2】 设图灵机纸带状态如图 1-16 所示。图中，x 和 y 为输入带符，B 为空格符。读写头的动作包括向左移动 1 格、向右移动 1 格、停止 3 个动作，并用 IN 和 OUT 来分别表示该图灵机的输入信息集合和输出信息集合，即

IN＝{x，y，B}

OUT＝{Left，Right，Stop}

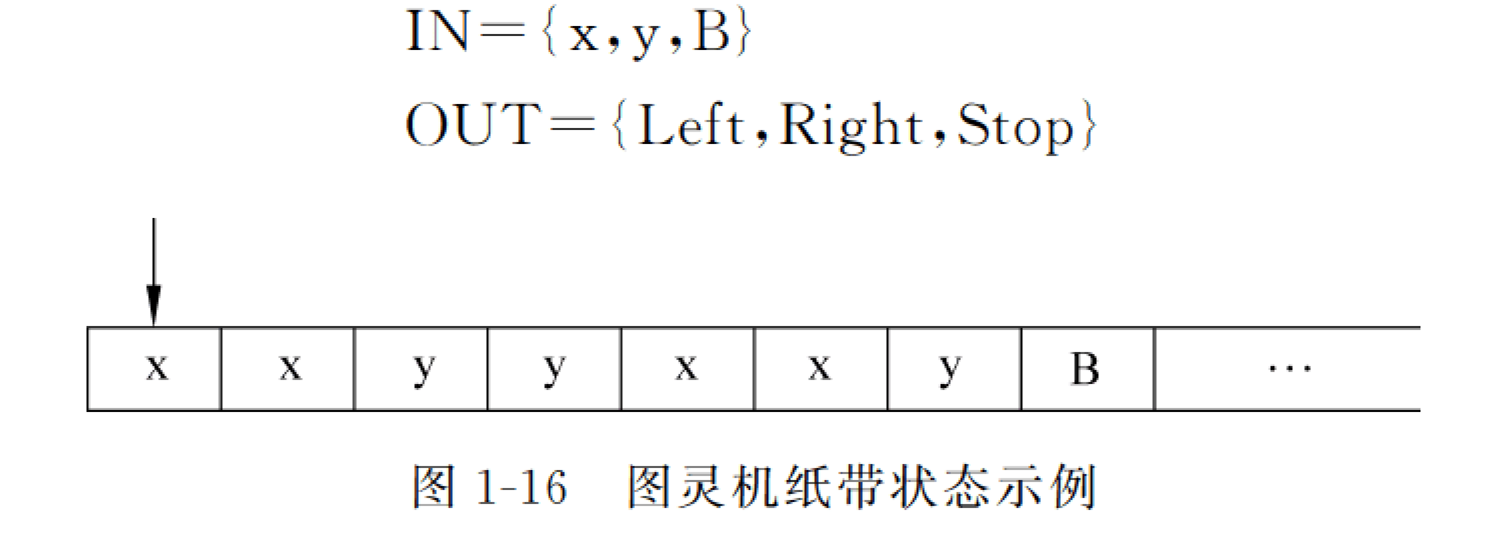


图 1-16　图灵机纸带状态示例

给定控制规则为[①]

① 从最左端起始。
② 若读入 x，右移一格，并仍为读状态。
③ 若读入 y，将其改为 x，再继续右移一格，并仍为读状态。
④ 若读入 B，停止。

依据上述规则及图 1-16 所示的纸带状态，读写头从最左端开始，移动 7 次后，纸带上停止符 B 左端的符号均改为 x，并停止。

① 控制规则表用于表示图灵机对输入应给出的响应输出。即，根据读入的带符及当前的内部状态，确定读写头的动作(移动方向)，以及是否改写当前带符。

由上例可得，图灵机的工作过程可以直观地描述为如下几个步骤：

（1）读写头从指向的纸带单元格中读出一个信息。

（2）根据当前的内部状态查规则表（table）。

（3）由规则表（程序）得出下一步要做的动作。即确定是否改写当前单元格中的信息，读写头是否移动、移动方向等。同时，由规则表还会得出下一时刻内部状态将会发生什么样的变化。

表 1-1 为规则表的一种形式。

表 1-1 规则表示例

当前内部状态	输入信息	输出动作	下一时刻的内部状态
A	1	前移	C
C	0	向纸带上写入 1	B
B	1	后移	A
⋮	⋮	⋮	⋮

图灵机只要根据每一时刻读写头读到的信息和当前的内部状态查规则表，就可以得出它下一时刻的内部状态和输出动作。这个模型看上去似乎很简单，但可以想到，只要改变规则表中的规则，图灵机的工作也就改变了。规则表越复杂，功能越强大，图灵机的功能也就越强。

到此，是不是看上去已经有点计算机的意思了？事实上，如果把图灵机的控制规则解释为指令，将规则集合解释为程序，都用二进制编码表示，与输出和输入信息（也可用二进制码表示）同样存储在机器里，编写不同的程序就会使机器做不同的动作，就成为电子计算机了（计算机所做的各种复杂工作就是靠程序去实现的）。有关图灵机与计算机的关系将在第 3 章中讨论。

3. 图灵机的形式化描述

图灵机（TM）可以描述为一个五元组①：

$$M = (Q, \Sigma, \delta, B, H) \tag{1.2}$$

其中：

Q 为图灵机状态的有穷集合。

Σ 为带符的有穷集合。

δ 为控制规则集合。

B 为初始状态，属于 Q，开始时图灵机就处于 q_0 状态。

H 为停机状态，是 Q 的子集（$H \in Q$）。当控制器内部状态为停机状态时，图灵机结束计算。

① 图灵机模型也可以描述为一个七元组。在七元组描述中，输入符号和带符分别用两个元素描述，并将空格符作为一个独立的元素出现。

1.2.2 图灵机构造示例

【例 1-3】 设计一个计算 $X+1$ 的图灵机，并在计算结束后读写头回到右端的起始位置。

为简单起见，这里假设 $X=5$，即设计一个计算 5+1 的图灵机。

设计思路：根据图灵机的形式化描述，需要设计图灵机的带符（输入）集合 Σ、内部状态集合（包括初始状态和停止状态）Q 以及控制规则 δ。

由于所设计的图灵机要完成的是计算 5+1，按二进制计数制考虑，设计其输入集合为

$$\Sigma=\{0,1,*\}$$

这里，* 是起始位置的标示符。

作为实现加法运算的图灵机，其内部状态集合应包括完成加法运算可能出现的各种状态。如求和、进位、溢出等。另外，图灵机还要求必须有起始和停止状态。因此，设计该图灵机的内部状态集合 Q 为

$$Q=\{\text{start},\text{add},\text{carry},\text{noncarry},\text{overflow},\text{return},\text{halt}\}$$

其中，各状态的含义分别为：start，起始；add，加；carry，进位；noncarry，无进位；overflow，溢出；return，返回；halt，停止。

最后，根据题目要求，定义控制规则，如表 1-2 所示。

表 1-2 控制规则

规则序号	输入		响应输出		
	当前状态	当前符号	新符号	读写头移动	新状态
1	start	*	*	left	add
2	add	0	1	left	noncarry
3	add	1	0	left	carry
4	add	*	*	right	halt
5	carry	0	1	left	noncarry
6	carry	1	0	left	carry
7	carry	*	1	left	overflow
8	noncarry	0	0	left	noncarry
9	noncarry	1	1	left	noncarry
10	noncarry	*	*	right	return
11	overflow	0 或 1	*	right	return
12	return	0	0	right	return
13	return	1	1	right	return
14	return	*	*	stay	halt

设初始情况下，纸带上的内容为5，读写头指向左侧的起始位置start（如图1-17所示）。

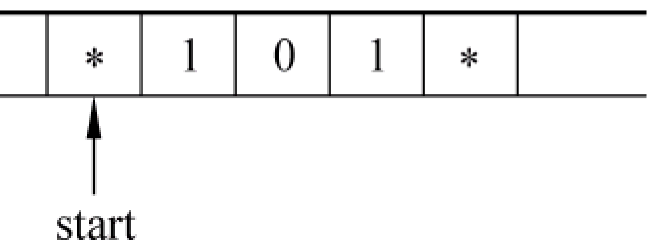


图1-17　图灵机初始情况

根据表1-2所示的规则表，该图灵机的工作过程如下：

（1）按照规则1，若当前状态为start，当前带符为*，则单元格中的带符保持为*，读写头向左移动一格，做“加”运算（如图1-18(a)所示）。

（2）由图1-18(a)查规则表，若当前状态是add，输入带符是1，则按规则3，将当前带符变为0，读写头向左移动一格，工作状态为carry（如图1-18(b)所示）。

（3）图1-18(b)符合规则5的输入，故由规则5得到读写头的下一步动作为：将当前带符改为1，读写头向左移动一格，工作状态为noncarry（如图1-18(c)所示）。

（4）同理，图1-18(c)符合规则9的输入条件。则由规则9，可得下一步工作状态为图1-18(d)所示。

（5）继续查规则表，发现图1-18(d)所示的状态符合规则10，则执行规则10，图灵机读写头向右返回到起始处（如图1-18(e)所示）。

（6）规则表中最后一条规则是：若当前状态为Return，带符为*，则读写头不动(stay)，状态变换为halt，即停止状态。图1-18(e)中读写头的状态符合规则14，故此时图灵机停止工作，呈现为图1-18(f)的形式。

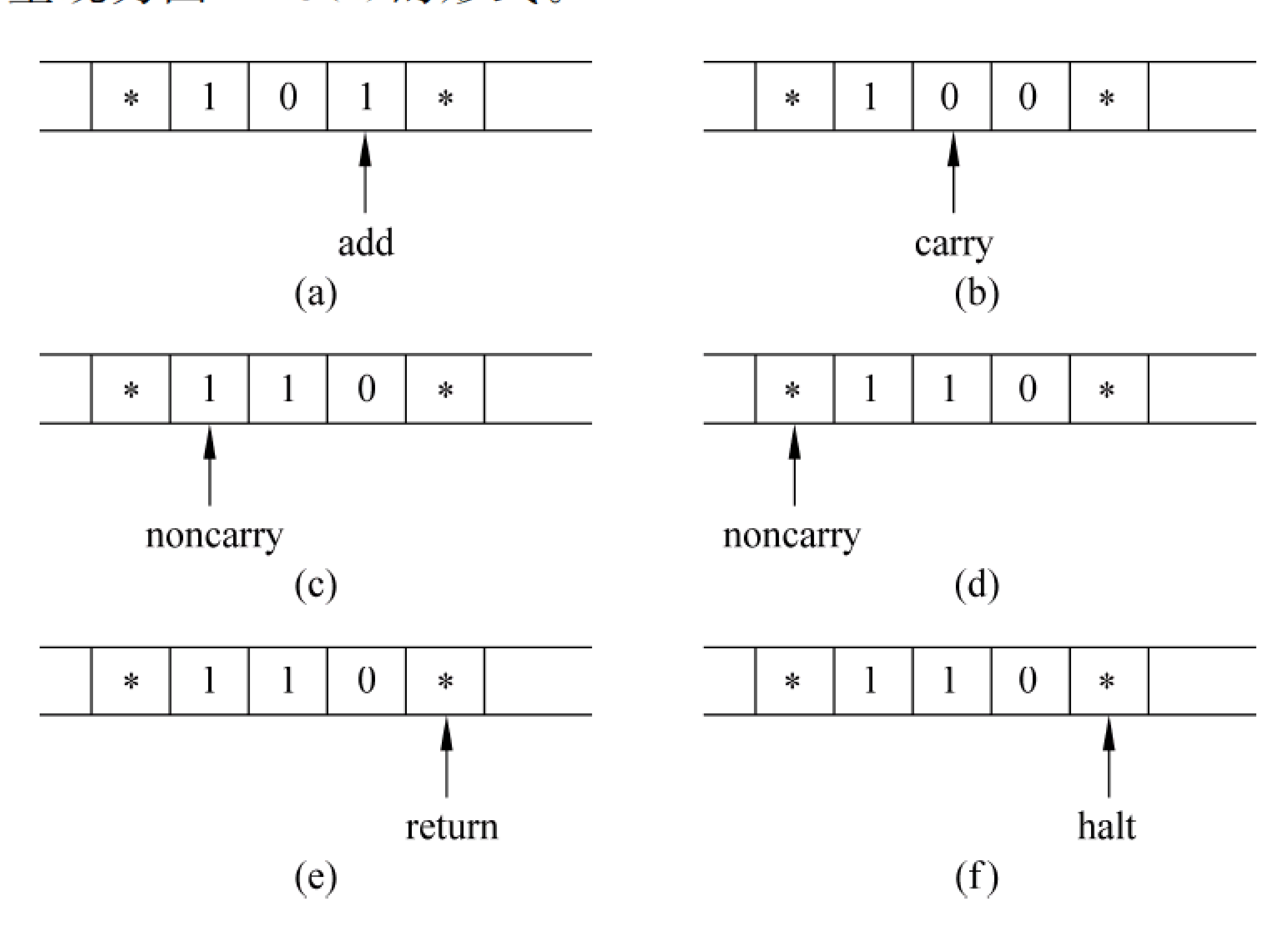


图1-18　图灵机的工作过程

由此，这个图灵机就完成了5+1的计算，纸带上的内容变为6。

当然，这个计算5+1的图灵机模型看上去很简单，似乎不足以让人感觉到图灵机模型的伟大。下面从另一个角度来看一下。从图灵机的结构及例1-3所示的工作过程，可以看出，图灵机模型无非包括4个要素：输入信息的集合（如表1-2中的“输入”栏）、输出信息的集合（如表1-2中的“响应输出”栏）、内部状态集合和一套固定的规则。图灵机的整个工作都依赖于这4个要素。如果修改图灵机的规则表，它的工作结果就会发生相应的变化，也会有不同的结果。

这样看来，图灵机的威力就可以很大了。因为大千世界中的很多事例，甚至包括人，

都可以抽象为图灵机模型。试想,如果将一个人每天的耳闻目睹看作输入集合,将人的一言一行看作输出集合,将人大脑中所有神经细胞的状态组合视为内部状态,而将人根据耳闻目睹所做出的反应(也就对每种输入信息的反应)认为是执行了某个程序,那么,会思考、会决策的人也就可以抽象为图灵机了。所以,图灵认定,人脑不会超越图灵机模型,也由此认为,人工智能是可能的(当然,这个问题已超出了本书的讨论范围)。

如果说人也可以抽象为图灵机,那图灵机的威力就很大了。那么,图灵机有能力极限吗?当然,世界上没有万能的东西,图灵机也不是例外,它也有它的能力极限。为了帮助读者初步了解图灵机的能力局限性(也就是计算机的能力局限性),下面先简单地讨论什么是计算。

1.2.3 计算与可计算性理论

1. 计算

计算的行为由来已久,考古研究表明,在远古时代,古人类就有了计算问题的需要和能力。人类最初的计算工具就是人类的双手,掰着指头数数就是最早的计算方法。一个人天生有十个指头,因此十进制就成为人们最熟悉的记数法。

由于双手的局限性,人类开始学习用小木棍、石子、结绳等方法进行计算。英文中的计算一词 Calculation 来自拉丁文中的 Calculus,其本意就是用于计算的小石子。古代中国人在两千多年前发明的算筹是世界上最早的计算工具,而具有十进制记数法和一整套计算口诀的算盘则可以认为是最早的"数字计算机",珠算口诀就是最早的体系化的算法。

但到底什么是"计算"?这个问题直到 20 世纪 30 年代人们才从哥德尔(K. Godel)、丘奇(A. Church)、图灵(A. M. Turing)等科学家的研究中弄清楚是计算的本质,更重要的是弄清楚了什么问题是可计算的,而什么问题是不可计算的。

所谓计算,抽象地讲,就是从一个符号串 X 变成另一个符号串 Y 的过程。例如:

(1) 从符号串 $5+8-4$ 变换为 9,是进行了加减计算。

(2) 将 $X=a^2$ 变换为 $Y=2a$,是进行了微分计算。

(3) 将一段英文(符号串 X)翻译为中文(符号串 Y),也是进行了计算。

从以上这几个简单示例可以看出,计算是按照一定的、有限的规则和步骤(算法),将输入转换为输出的过程。

从数学的角度,计算主要可以分为数值计算和符号推导两大类。数值计算包括各种算术运算、方程求解等,如上例的(1)和(2);符号推导包括各种函数的等式或不等式的证明、几何命题证明等,上例中的(3)也可以归为符号推导。

2. 计算科学

伴随着计算机技术的发展,计算这个原本属于专业领域的数学概念已经泛化到人类的整个知识领域,并成为一种普适的科学概念。计算不再只是数学的基础技能,而是整个自然科学的工具。今天,人类的研究领域越来越广,每个领域的研究都会涉及大量的计

算。作为各门学科研究的基础，在现代计算工具的支撑下，计算也逐渐形成体系，成为一门独立的学科，即计算科学，它是涉及数学模型构建、数值分析方法以及计算机实现方法的研究领域。

数学模型(mathematical model)是用数学符号、数学公式、程序、图形等对客观问题本质属性的抽象、简洁的刻画，它或能解释某些客观现象，或能预测未来的发展规律，或能为控制某一现象的发展提供某种意义下的最优策略或较好策略。数学模型的建立首先需要对现实问题进行深入细致的观察和分析，抽象出各种关键因素并确立它们之间的关系，再灵活巧妙地利用各种数学知识，表述这些关系。例如，学生综合素质测评、股市变化分析、气象综合预测等问题都可以建立一个数学模型。通过研究对象特征，分析对象的内在规律，确定影响变化规律的因素，利用适当的数学工具，构造出各因素间的关系。

这种应用相关领域知识从客观问题中抽象、提炼出数学模型的过程就称为数学建模(mathematical modeling)。数学建模的核心是分析和抽象，即从复杂的客观现实中抽取出反映其变化规律的各种因素，并建立它们之间的关系。因此，建立数学模型的目的就是为了便于分析和研究各种客观现象的运动规律，从而确立最佳的问题解决方案。

数值分析(numerical analysis)是关于数值近似算法的研究。古巴比伦人曾利用巴比伦泥板(数值分析的最早作品之一)来计算$\sqrt{2}$的近似值，而不是精确值。引入数值分析的原因是因为在许多实际问题中，常常无法求得精确值，或是无法用有理数表示结果，比如$\sqrt{3}$。数值分析的目的就是在合理的误差范围内求得满足精度要求的近似结果。数值分析方法在所有工程和科学领域中得到应用，例如数值天气预报、太空船的轨迹计算、股票市值及其变异程度分析、保险公司的精算分析等。

计算机实现方法可以描述为利用计算机求解问题的方法。在模型建立的前提下，“计算机实现”的核心就是算法设计和程序设计。

虽然计算科学不完全等同于计算机科学，但计算科学以及其他各学科的研究都离不开计算机。因此，利用计算机分析和解决相关问题的能力成为每一位科学工作者应具备的基本素质。这种能力也称为计算思维(computational thinking)[①]能力。

计算机科学(computer science)是主要研究计算理论、计算机及信息处理的学科。半个多世纪来，计算机科学得到飞速发展和普及，作为现代科学体系的主要基石之一，它已逐渐超出一门单独学科的范围，演变为一种与社会、经济、能源、材料、健康等多个领域相结合的横向型科学技术。

有关计算机科学的详细定义有多种，但不论哪一种定义，都强调了算法的研究。算法描述了解决某个特定问题的确切、无歧义、有限的动作序列。如按照某些准则获取一个年级中综合成绩最好的学生的过程，就是一种算法；按照菜谱一步步做出一道好菜的过程，也是一种算法。计算机科学既要研究算法分析与设计理论，也同时要考虑如何在计算机上实现算法并解决实际问题。

---

① 计算思维是运用计算机科学的基础概念进行问题求解、系统设计以及人类行为理解等涵盖计算机科学之广度的一系列思维活动。

3. 可计算性理论

可计算性理论(computability theory)是研究计算的一般性质的数学理论。由于计算的过程就是执行算法的过程,因此,可计算理论的中心课题就是将算法这一直观概念精确化,建立计算的数学模型,研究哪些是可计算的,哪些是不可计算的,以此揭示计算的实质。由于计算是与算法联系在一起的,因此,可计算性理论也称算法理论。

通常,人们把能用笔在纸上经有限步计算就可得到结果的问题称为可计算的问题。这里的"有限步"也可以直观地说成是一组可操作或可执行(用笔在纸上算就是在操作)的规则。这样一组条数有限、每一条都可执行的规则就可以称为算法。这里的可执行性是绝对机械的,即不论何人何时对之进行操作,只要输入数据相同,其结果都是一样的。由于条数有限,所以作为算法的一组规则中至少包含一条终止计算的规则。因此,从直观上看,算法具备的特征是有限性、可执行性、机械性、确定性和终止性。

在 20 世纪以前,对"算法"、"计算"这些概念似乎并不存在什么问题,人们普遍认为所有的问题都是有算法的,至少是一切数学命题都存在算法。但是 20 世纪初,人们发现有许多问题虽已经过长期研究,仍然找不到算法。如希尔伯特第 10 问题、半群的字的问题等(这些概念的描述需占较大篇幅,已超出本书的讨论范围,请读者自行查阅其他相关资料)。于是人们开始怀疑,是否对某些问题根本就不存在算法?即是否存在不可计算的问题?

数学家们由此开始了对算法概念及可计算性的精确化研究。1934 年,哥德尔(Godel)提出了一般递归函数的概念,并证明了一般递归函数是算法可计算函数①,反之亦然。也就是说,一般递归函数是可计算的。同年,丘奇证明了他提出的 λ 可定义函数与一般递归函数是等价的,并提出算法可计算函数等同于一般递归函数或 λ 可定义函数,即著名的"丘奇论题"。图灵则在他的《论可计算数及其在判定问题中的应用》及《可计算性与 λ 可定义性》等论文中,以图灵机模型为基础,证明了图灵可计算函数②与 λ 可定义函数是等价的,也就证明了**能用图灵机计算的函数是算法可计算函数**(即这样的函数是可计算的),而图灵机不能计算的函数则是不可计算的函数。这一结论(就是著名的"丘奇-图灵论题")相当完善地解决了可计算函数的精确定义问题,对数理逻辑的发展起了巨大的推动作用。

虽然已经证明可计算函数就是图灵机可计算函数,而图灵机可计算函数是递归可枚举函数。但这样的定义或描述对初学者来讲显得过于高深。事实上,由于图灵机与计算机可以相互模拟(参阅 3.4.3 节),所以,对"可计算性"问题也可以通俗地描述为:可计算的问题就是可以用计算机来解决的问题。从广义上讲,如"请为我做一个汉堡"这样的问题是无法用计算机来解决的(至少目前还不行)。计算机本身的优势在于数值计算(很多如文字识别、图像处理等非数值问题都可以转化为数值问题),**能够用计算机解决的问题**

---

① 可计算函数定义为:能够在抽象计算机上编出程序计算其值的函数。

② 若设 $f(x_1,x_2,\cdots,x_n)$ 是定义在自然数集上的函数,如果存在图灵机 M_f,对任意输入的 $x_1,x_2,\cdots,x_n$,当有定义 $f(x_1,x_2,\cdots,x_n)=y$ 时,M_f 执行有限步终止并输出 y,则 $f(x_1,x_2,\cdots,x_n)$ 为图灵可计算函数。

一定是“可以在确定的有限步骤内被解决的问题”，即有确定算法。像哥德巴赫猜想这样的问题就不属于可计算问题之列，因为计算机没有办法给出数学意义上的证明。

有关可计算性理论的深入描述因篇幅及课程性质所限，不再做进一步的讨论。本节讨论可计算性问题的目的，是希望能使读者能初步了解：**计算机不可能解决世界上所有的问题。其“不可解决性”反映在两个方面，一是不能在有限步骤内被解决；二是虽然有可能解决，但因过于复杂而不能在可接受的时间内解决**（所有机械装置都存在复杂性的临界点）。关于后者可参阅7.3节中关于算法复杂性评价的内容。

分析某个问题的可计算性意义重大，它可以使人们不必浪费时间在不可能解决的问题上，而将精力用于可以解决的问题集中。

图灵机的产生，一方面奠定了现代数字计算机的基础（后来的冯·诺依曼就是根据图灵的设想设计出第一台计算机的）；更重要的是说明了什么样的问题是可计算的，而什么样的问题是不可以计算的。亦即说明了可计算的极限（即所谓的图灵停机问题①）是什么。这也为今天计算机的能力极限给出了结论。

作为学习计算机科学的入门教材，首先使读者了解“不是任何问题都是计算机可解的”这一概念是非常有必要的。虽然在现代科学研究中没有计算机是万万不能的，但计算机确实不是万能的。

*1.3 计算工具的发展与启示

计算工具是用于完成计算的器具。自有数字诞生那天起，计算就开始存在，计算工具也就开始伴随着人类进化和发展。从远古到今天，计算工具由最初的绳结、手指、石块等，到之后的算筹、算盘、计算尺，再发展到各种机械式计算机，最后到今天的电子计算机。人类在发展过程中，不断地寻找和探索着更高效、更适用的计算工具，为的就是希望能更快速地解决越来越复杂的各种计算问题。

对于电子计算机诞生之前的各类计算工具，已有多种书籍或网络文章做过详细介绍，限于篇幅，本节仅简单介绍电子计算机的发展历程，并由此讨论未来计算工具可能的方向。

1.3.1 电子计算机的诞生和发展

在1946年之前，计算机的工作都是基于机械运行方式，没有进入逻辑运算领域。如果不是1906年美国人Lee De Forest发明了电子管，电子计算机是不可能出现的。正是电子技术的飞速发展，使计算机由机械式发展到电子时代。

计算机的发展至今经历了5个时代。第一代（1946—1954）称为“电子管计算机”时

① 停机问题是指：是否存在一个算法，对于任意给定的图灵机都能判定任意的输入是否会导致停机？已证明图灵机的停机问题是不可判定的。停机问题的不可判定性成为解决许多不可判定性问题的基础。

代，内部元件使用电子管。图 1-19 所示是第一台用电子管和继电器制作的通用电子计算机 ENIAC，它于 1946 年 2 月 15 日在美国费城大学问世，共使用了 18 800 个电子管，6000 多个开关和配线盘，重约 30 吨，占地 1500 平方英尺，工作主频为 0.1MHz。每当进行不同的计算时，都需要切换开关和改变配线，这使当时从事计算的科学家看上去更像在干体力活。

美国数学家冯·诺依曼提出了解决此问题之道，这就是“程序存储方式”。通俗地讲就是把原来通过切换开关和改变配线来控制的运算步骤，以程序方式预先存放在计算机中，然后让其自动计算。在以后的日子中，计算机的发展正是沿着“程序存储方式”这一光辉道路前进的。

但无论如何，它的诞生表示人类从此进入了电子计算机时代。从那一天至今的半个多世纪中，随着电子技术的发展，计算机经历了电子管、晶体管、集成电路、大规模集成电路以及超大规模集成电路等几代的发展，无论是在体积上、运行速度上，还是在智能性、可靠性以及价格等多方面都有了迅猛的进步，成为 20 世纪发展最快的技术，计算机行业也成为 20 世纪最具活力的行业。

第一代计算机主要用于工程计算，其主要特点是采用电子真空管和继电器构成处理器和存储器，用绝缘导线实现互连。体积较为庞大，运算速度较低，运算能力有限。程序编写采用由 0 和 1 组成的二进制码表示的机器语言，只能进行定点数运算。由于电子管易发热，寿命最长只有 3000 小时。因此计算机运行时常会因电子管被烧坏而出现死机。

第二代计算机属于晶体管计算机（1960—1964）。晶体管（transistor）是一种半导体器件，具有检波、整流、放大、开关、稳压、信号调制等多种功能，可以作为一种开关元件。与普通机械开关不同，晶体管利用电信号来控制自身的开合，其开关速度可以非常快（实验室中的切换速度可达 100GHz 以上）。图 1-20 所示为第一台全晶体管计算机，它共装有 800 只晶体管。

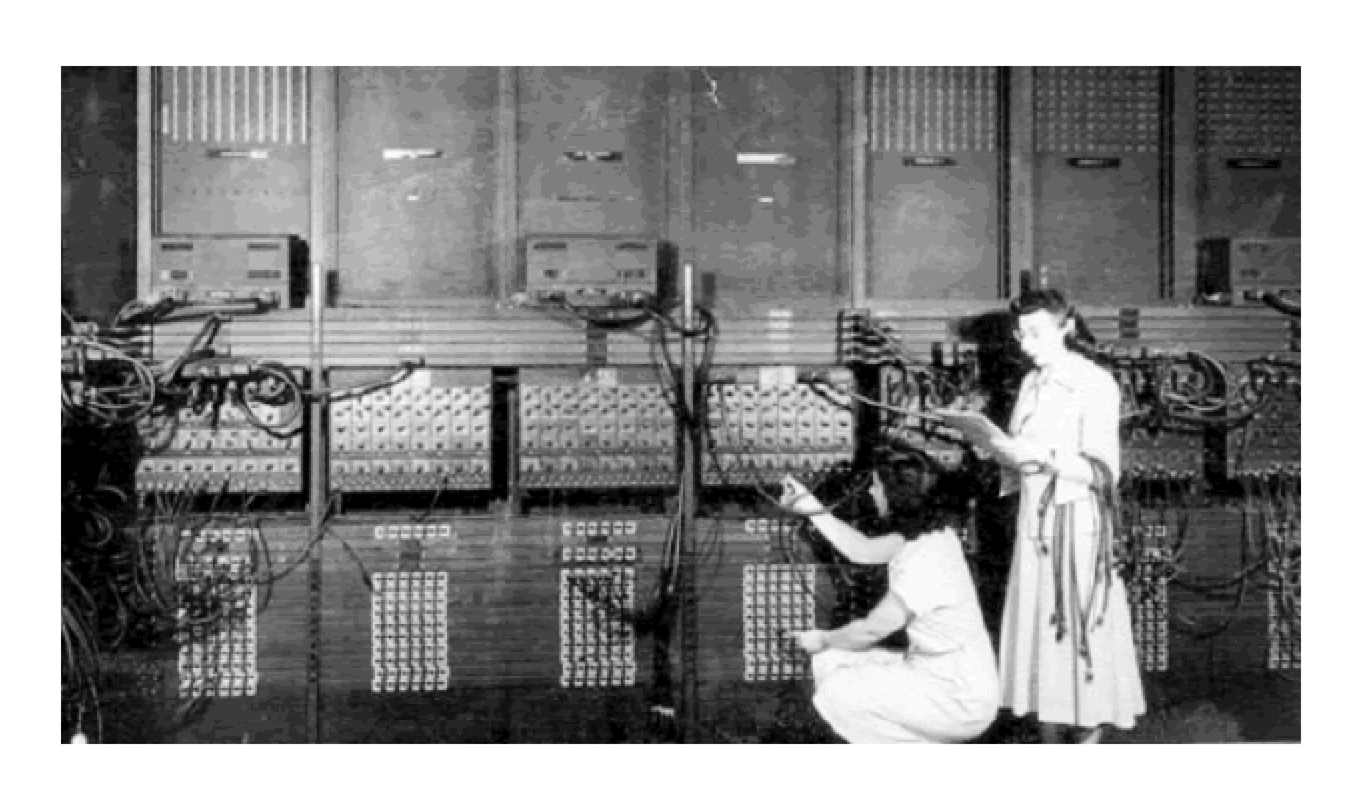


图 1-19　在第一台电子计算机 ENIAC 上编程



图 1-20　TRADIC 晶体管计算机

第二代计算机采用晶体管逻辑元件及快速磁芯存储器，彻底改变了继电器存储器的工作方式和与处理器的连接方法，大大缩小了体积。其运算速度也从第一代的每秒几千次提高到几十万次，主存储器的存储容量从几千字节提高到 10 万字节以上。另外，第二代计算机普遍增加了浮点运算，使数据的绝对值可达到 2 的几十次方或几百次方，同时有了专门用于处理外部数据输入输出的处理机，使计算能力实现了一次飞跃。计算机除科

学计算外，开始被用于企业商务。

在软件方面，第二代计算机除机器语言外，开始采用有编译程序的汇编语言和高级语言，建立了子程序库及批处理监控程序，使程序的设计和编写效率大为提高。

采用集成电路作为逻辑元件是第三代计算机(1964—1974)的最重要特征。此时，微程序控制、流水线技术、高速缓存和先行处理机等技术开始出现并逐渐普及。第三代计算机的典型代表有1964年IBM公司研制出的IBM S/360、CDC公司的CDC6600及Cray公司的巨型计算机Cray-1等(如图1-21所示)。



图1-21　Cray-1巨型计算机

随着集成电路技术的发展，出现了采用大规模和超大规模集成电路及半导体存储器的第四代计算机(1974—1991)，同时，计算机也逐渐开始依据功能和性能的不同分为巨型机、大型机、小型机和微型机。出现了共享存储器、分布存储器及不同结构的并行计算机，并相应产生了用于并行处理和分布处理的软件工具和环境。第四代计算机的代表机型Cray-2和Cray-3巨型机，因采用并行结构而使运算速度分别达到12亿次每秒和160亿次每秒。

从1991年至今的计算机系统，都可以认为是第五代计算机。超大规模集成电路(VLSI)工艺的日趋完善，使生产更高密度、高速度的处理器和存储器芯片成为可能。这一代计算机的主要特点是大规模并行数据处理、系统结构的可扩展性、高性能的实时通信能力和智能性。随着集成电路技术的不断发展，现代计算机系统的运算速度和整体性能都得到不断提高。图1-2是我国2013年推出的天河二号超级计算机，其峰值计算速度可以达到5.49亿亿次每秒。

1.3.2　微型计算机的发展

相对于高性能大型或巨型计算机系统，在20世纪70年代诞生的微型计算机(也称PC，Personal Computer，个人计算机)则因其较高的性价比而在各行各业中得到了更为广泛的应用。

微型计算机的发展伴随的是微处理器的发展。世界上第一片微处理器是Intel公司1971年研制生产的Intel 4004(如图1-22所示)，是一个4位微处理器，可进行4位二进制的并行运算，拥有45条指令，速度为0.05MIPS。

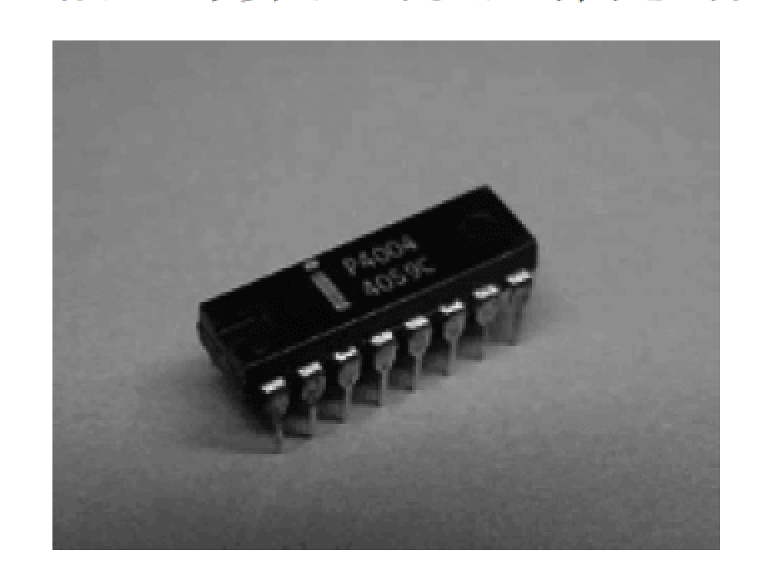


图1-22　Intel 4004微处理器芯片

Intel 4004功能有限，主要用于计算器、电动打字机、照相机、台秤、电视机等家用电器上，一般不适用于通用计算机。而在同年末推出的8位扩展型微处理器Intel 8008则是世界上第一片8位微处理器，也是真正

适用于通用微型计算机的处理器。它可一次处理8位二进制数，寻址16KB存储空间，拥有48条指令。这些使它能有机会应用于许多高级的系统。

微处理器及微型计算机从1971年至今经历4位、8位、16位、32位、64位及多核芯6个时代。除主要用于袖珍式计算器的4004芯片外，其他具有划时代意义微处理器有：

- 1973年Intel公司推出的8位微处理器Intel 8080。这是8位微处理器的典型代表。它的存储器寻址空间增加到64KB，并扩充了指令集，指令执行速度达到50万条指令每秒，同时它还使处理器外部电路的设计变得更加容易且成本降低。除Intel 8080外，同时期推出的还有Motorola公司的MC6800系列，以及Zilog公司的Z80等。
- 1978年推出的Intel 8086/8088微处理器是16位微处理器的标志。其内部包含29 000个3μm技术的晶体管，工作频率为4.77MHz，采用16位寄存器和16位数据总线，能够寻址1MB的内存储器。IBM PC采用的微处理器就是8088。同时代的还有Motorola公司的M68000和Zilog公司的Z8000。
- 1985年研制成功的32位微处理器80386系列。其内部包含27.5万个晶体管，工作频率为12.5MHz，后逐步提高到40MHz。可寻址4GB内存，并可管理64TB的虚拟存储空间。
- 奔腾(Pentium)微处理器在2000年11月发布，起步频率为1.5GHz，随后陆续推出了1.4～3.2GHz的64位的P4处理器。
- 2006年开始推出并得到迅速发展的多核处理器，是计算技术的又一次重大飞跃。多核处理器是指在一个处理器上集成两个或以上运算核心，从而提高计算能力。较之单核处理器，多核处理器能带来更高的性能和生产力优势，因而成为一种广泛普及的计算模式。图1-23为Intel公司近年推出的新型多核处理器芯片。



图1-23　Intel Core i7 微处理器芯片

世界上第一台微型计算机Altair8800在1975年4月由Altair的公司推出，它采用Zilog公司的Z80芯片做微处理器。它没有显示器和键盘，面板上有指示灯和开关，给人的感觉更像一台仪器箱。

IBM公司在1981年推出了首台个人计算机IBM PC，1984年又推出了更先进的IBM PC/AT，它支持多任务、多用户，并增加了网络能力，可联网1000台PC。从此，IBM公司彻底确立了在微机领域的霸主地位。

今天，微型计算机已真正进入千家万户、各行各业，它在功能上、运算速度上都已超过了当年的大型机，而价格却只是大型机的几分之一，真正实现了其大众化、平民化和多功能化的设计目标。

1.3.3　未来计算机的发展

未来充满了变数，未来的计算机将会是什么样？

21 世纪是人类走向信息社会的世纪，是网络的时代，是超高速信息公路建设取得实质性进展并进入应用的年代。电子计算机技术正在向巨型化、微型化、网络化和智能化 4 个方向发展。

巨型化不是指计算机的体积大，而是指运算速度高、存储容量大、功能更完善的计算机系统。巨型机的应用范围如今也日渐广泛，在航空航天、军事工业、气象、电子、人工智能等几十个学科领域发挥着巨大的作用，特别是在复杂的大型科学计算领域，其他的机种难以与之抗衡。

计算机的微型化得益于大规模和超大规模集成电路的飞速发展。现代集成电路技术的发展，已可将计算机中的核心部件——运算器和控制器集成在一块大规模或超大规模集成电路芯片上，作为中央处理单元，称为微处理器，从而才使计算机作为“个人计算机”变得可能。微处理器自 1971 年问世以来，发展非常迅速，伴随着集成电路技术的发展，以微处理器为核心的微型计算机的性能不断跃升。现在，除了放在办公桌上的台式微型机外，还有可随身携带的各种规格的笔记本计算机、可以握在手上的掌上电脑、可随时上网和进行文字处理的平板电脑、手机等。

据美国媒体报道，在 2011 年 2 月，美国科学家已成功研制出世界上最小的计算机——一种可以植入眼球的医用毫米级计算系统。这种计算机主要为青光眼患者研制，放置在患者眼球内，可以监测眼压，方便医生及时为病人缓解痛苦。据介绍，这种计算机只有一立方毫米大小，包括一个极其节能的微处理器、一个压力传感器、一个记忆卡、一块太阳能电池、一片薄薄的蓄电池和一个无线收发装置。通过无线收发装置，这个计算机能够向外部装置发出眼压数据资料。

从 20 世纪中后期开始，网络技术得到快速发展，已经突破了只是“帮助计算机主机完成与终端通信”这一概念。众多计算机通过相互连接，形成了一个规模庞大、功能多样的网络系统，从而实现信息的相互传递和资源共享。今天，网络技术已经从计算机技术的配角地位上升到与计算机技术紧密结合、不可分割的地位。各种基于网络的计算机技术不断出现和发展(参见 1.4 节)，计算机连入网络已经如同电话机连入市内电话交换网一样方便，且网络信息传送的速度也随着“光纤”差不多铺到“家门口”而变得越来越快。今天，计算机技术的发展已离不开网络技术的发展，同时，网络也成为人类生活的一部分。

计算机的智能化就是要求计算机具有人的智能，即让计算机能够进行图像识别、定理证明、研究学习、探索、联想、启发和理解人的语言等。目前人工智能技术的研究已取得较大成绩，智能计算机(俗称“机器人”)已部分具有人的能力，能具有简单的“说”“看”“听”“做”能力，能替代人类去做一些体力劳动或从事一些危险的工作。如日本福岛核电站出现核泄漏后，日本政府就曾“派”机器人进入核电站检测核泄漏的情况。

人工智能是目前以至未来可见的时间里计算机科学的研究热点。人工神经网络的研究，使计算机向人类大脑接近又迈出了重要的一步。今天，除了希望在软件技术上的不断深入研究，人们还寄希望于全新的计算机技术能够带动人工智能的发展。至少有 3 种技术有可能引发全新的革命，它们是光子计算机、生物计算机和量子计算机。

光子计算机的运算速度据推测可能比现行的超级计算机快 1000 到 1 万倍。而一台具有 5000 个左右量子位的量子计算机可以在大约 30 秒内解决传统超级计算机需要 100

亿年才能解决的素数问题。相对而言，生物计算机研究更加现实，美国威斯康星-麦迪逊大学已研制出一台可进行较复杂运算的 DNA 计算机。据悉，一克 DNA 所能存储的信息量可与 1 万亿张 CD 光盘相当。这些推测，使人们对人工智能的发展前景变得乐观。

思考 计算机真的能达到人类的思维能力、模拟人类的行为动作吗？未来的计算机会像影视剧中描述的那样完全达到人的智力吗？

*1.4 基于计算机的问题求解

计算机学科要解决的根本问题就是“利用计算机进行问题求解”。因此，有必要首先了解利用计算机进行问题求解的一般过程。

日常生活中，每个人每天都会遇到大大小小的各种问题。遇到问题就需要解决问题。那么，我们是否总结过在遇到问题时是怎样建立起解决问题的思路，又是怎样选择了解决问题的方法呢？

事实上，每个人在遇到问题时，都有意识或无意识地经历了以下这样的过程：

(1) 对问题是否可解决做一个可能是快速的评估，即该问题是否可解决。

(2) 确定解决问题的方法。对一个简单的问题，可能不需要过多地思考就能立刻解决。但如果是一个大问题(系统性问题)，可能就需要对问题进行分解，分配给多个人、花费一定的时间去完成。

(3) 每个领到任务的人会确定解决这个具体问题的方法并完成它。

(4) 当每个人分配到的“子问题”都解决之后，需要将所有的结果汇总在一起，以构成对大问题的解决结果。为了保证能够将“答案”合成到一起，需要在分配任务时就要说清楚提交“答案”的格式。

(5) 确认问题解决的正确性。

这是人类对大问题求解的过程，也是计算机求解系统性问题的一般过程。假设某大学要开发一套利用计算机进行控制的课程管理系统。对这个问题，需要做哪些工作呢？

作为计算机专业人员，可能对学校的学生管理模式不了解。因此首先需要了解这个系统应该有些什么功能，这需要和学校有关人员进行沟通。但学校的管理者对计算机技术可能不熟悉，他们不清楚哪些是计算机可以或容易实现的，而哪些是很难以实现的(计算机不是万能的)。因此，这样的沟通可能需要若干次，沟通的最终目的是要弄清楚用户的需求。所以，这个过程被称为需求分析。

在需求弄清楚之后，就进入了系统的设计阶段。它包括对功能模块的划分、算法设计、程序编写和调试以及最后的系统测试。或许在设计甚至程序编码阶段，会突然发现当初对用户的需求没有正确理解，抑或用户需求又发生了变化，那么就需要重新修改需求，并依次修改后续的设计、编码等。

理论上，上述这个过程中的一个步骤结束后才可以开始下一个步骤，但它们又是可以反馈的。下一个步骤进行中可能会发现上一步存在某些不足或不完善处，此时就需要返

回去修改。对系统性问题的求解过程可以用如图 1-24 所示的模型表示。

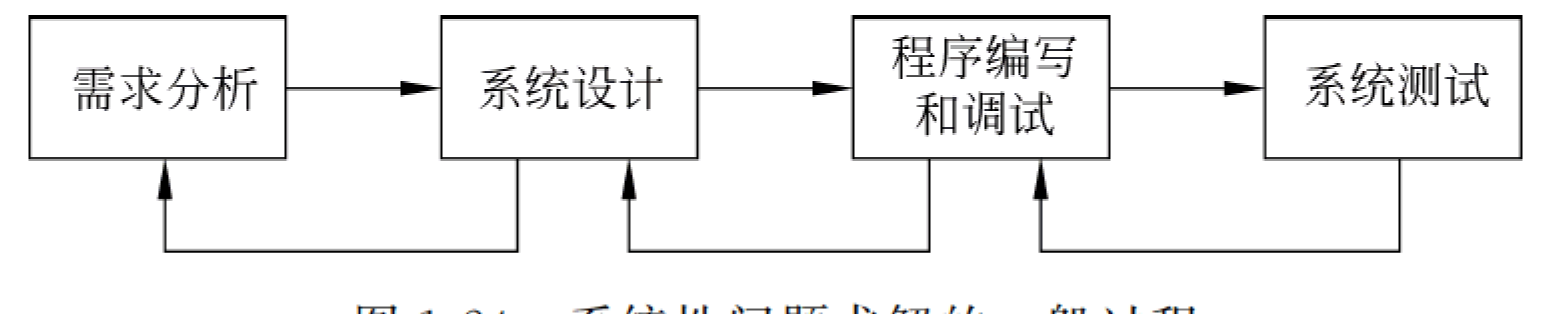


图 1-24　系统性问题求解的一般过程

1.4.1　需求分析与模型建立

需求分析的主要任务是：在充分理解用户需求（如课程管理系统需要有哪些功能）及对目标系统（如课程管理系统）领域知识有一定了解的基础上，确定问题“是否可解决”及“解决什么”。

“是否可解决”即解决问题的可行性。包括计算机科学求解问题的局限性、现有人员的技术水平、可能存在的经济和技术风险等；而“解决什么”则是在了解了问题所具有的特征、特点等基础上，对问题进行抽象，从而建立起系统模型。

从物理域中弄清楚要解决的问题，并通过对问题的抽象建立起逻辑模型，是一个复杂的过程，主要存在以下几个难点：

（1）问题的复杂性。用户需求涉及的因素繁多，如运行环境和系统功能等，引起问题的复杂性。

（2）交流障碍。需求分析涉及人员较多，这些人具备不同的背景知识，处于不同角度，扮演不同角色，造成相互之间交流困难。

（3）不完备性和不一致性。用户对问题的陈述往往是不完备的，各方面的需求可能还存在矛盾，需求分析要消除矛盾，形成完备及一致的定义。

（4）需求易变性。把握需求不是一蹴而就的，因此变化是客观的。

需求分析的主要工作有以下几个：

（1）问题识别。要确定用户对目标系统的综合要求（这里的目标系统就是按照用户需求，最终由计算机软件实现的系统），提出这些需求实现的条件以及最后应达到的标准。包括功能、性能、可靠性、用户界面等。

（2）分析建模。通过对问题的分析和方案的综合，逐步细化和明确目标系统的各项功能，建立问题求解的模型。

（3）编写需求分析文档。包括编写“需求规格说明书”“初步用户使用手册”“确认测试计划”等。

需要说明的一点是：通常所说的计算机软件并不仅指程序，还包括在整个问题求解过程中所编写的各种相关文档资料（这是相当重要的）。即计算机软件是程序、数据以及各种相关文档资料的总和。上述的需求分析文档就是这些文档中的一部分。

总之，需求分析的主要目标就是建立起系统的逻辑模型。所谓模型，是指对某一真实系统（如课程管理系统、学籍管理系统等）的目标、结构、行为等的抽象描述。模型的内容一般包括对象（概念）和对象之间的关系。建模的过程就是识别概念和概念间的关系，并

利用对象、联系等基本模型元素来描述系统的结构和行为等。

建模的根本是抽象，抽象是将问题域中的各种人、物、人或物之间的联系抽取出来，用一些特有的符号或形式表示。简单地讲，就是对欲求解问题给出清晰的定义和描述。比如，对于课程管理系统，系统中涉及了哪些对象？它们都有哪些属性？这些对象间有什么样的关系？有什么样的输入和输出？系统需要处理哪些信息及做什么样的加工等。这些问题看上去很玄，但利用课程管理这个简单系统，读者也许就能初步地理解。

经过对课程管理系统的需求分析，可以确定出系统中应包含的对象有教师、学生和课程。每个对象都有它们的属性。如学生的属性有学号、姓名、性别、年龄、班级等；课程的属性可以有课程代号、课程名、学分、学时等。教师和学生的关系可以是一位教师带多位学生，称为一对多关系。系统的输入就是已知什么条件，比如学生姓名、学号、选课代号、考试成绩；系统的输出就是希望得到什么结果，比如在屏幕上显示出某班选修某门课程的所有学生的姓名和考试成绩等。系统所做的处理就是希望计算机对输入信息做什么加工，比如对某班选修某门课程的所有学生的成绩进行排序，并统计成绩为优秀的学生人数等。

课程管理系统是一个相对比较简单的系统，当需要解决的问题比较复杂时，对问题的定义也就会变得非常复杂。这时需要借助于一些原则、方法和工具。有兴趣的读者可参阅有关软件过程学方面的书籍。

1.4.2 模块设计

需求分析阶段解决了要“做什么”的问题，设计阶段则是要解决“怎么做”。设计目标就是说明系统是如何被实现的。

对复杂的问题（大型系统），理解起来总是比较困难的，这时需要将大问题分解成若干小问题，以方便理解和解决，这就是系统的模块化。在分析阶段，已经建立了整个系统的模型，并对功能模块进行了大致的划分（层次划分），确定了系统的总体输入、处理和输出。但没有确定该怎么做。所以在设计阶段就需要解决以下几个问题：

（1）利用某种设计方法，将复杂问题划分为具体的子问题，即设计出应该包含哪些具体的功能模块。

（2）确定每个模块的功能。

（3）确定模块之间的关系、相互间应传递的信息。

（4）确定模块之间的联系方式（接口）。

（5）确定每个模块功能的实现方法和步骤（即算法）。

对于大多数软件系统，设计阶段还包括对数据结构和数据库的设计。当然，最后还需要编写设计文档。

读到这里，可以离开书本想一下，计算机能做什么？读者可能会想计算机能做很多很多事，但事实上，计算机只能做一件事，就是执行程序。它所能完成的每项工作都是通过执行程序来实现的，只是，不同的工作需要不同的实现方法，从而有不同的程序。所以，程序是建立在方法的基础上的。这里的方法，就称为算法。算法是实现某个具体功能的方法和步骤，是对问题处理过程的进一步细化。它不是计算机可以直接执行的，只是编写程

序代码前对处理思想的一种描述，它可以是自然语言，也可以是其他的方式。

有关算法分析的详细介绍将在第 7 章呈现给读者。下面仅通过一个实例来说明算法的描述。

【例 1-4】 给出以下问题的算法描述：统计某个班学生的英语和数学这两门课的成绩，找出合计成绩最高并且没有不及格课程的学生。

算法描述如下。

步骤 1：输入全部学生姓名、学号、英语成绩和数学成绩。

步骤 2：对各个学生的成绩求合计。

步骤 3：按合计对学生进行排序。

步骤 4：从排序列表中取第一位学生的成绩。

步骤 5：该学生有不及格课程吗？没有则打印姓名并结束。

若有不及格课程，则取下一位学生的成绩并重复步骤 5。

1.4.3 程序编码与调试

1. 编码

编码就是按照需求分析和模块设计所规划好的蓝本，用真正的计算机语言去实现所规划的功能。这一阶段主要涉及编码的组织及程序语言的选择。

1）自顶向下、逐步求精的设计方法

对于很小的简单的程序(比如十几行的程序)，程序怎么组织显得并不重要，但对于一个复杂的、成千上万行代码的大程序就不一样了(设想一下，连续 100 万行代码，如果连成一片该怎么管理)，需要将其自顶向下、一步步地划分为若干个小程序块，每一个小程序块(子程序，subprogram)完成相对单一的功能(这就像一个复杂的大问题需要逐级划分为若干子问题一样)，最终形成一个如图 1-25 所示的树状结构。

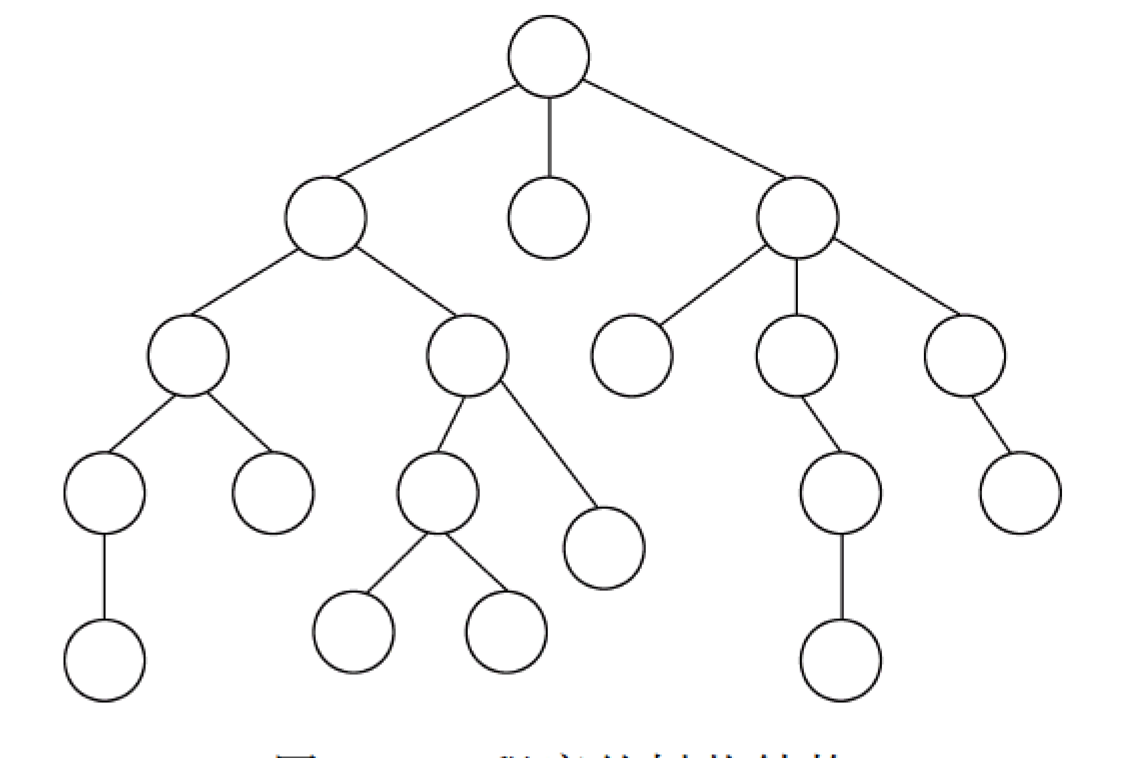


图 1-25　程序的树状结构

这种模块化的设计可以使程序的结构清晰，分工合作容易，编写和修改都比较方便。

2）程序设计语言的选择

目前可用的计算机语言有数百种之多，每种语言的功能和性能也在不断地改进。不同的语言有不同的特点和表现形式，同样的问题用不同的语言编写出的程序也会有不同，有时甚至会有较大的差别。有的程序语言适合数据库的开发，有的适合科学计算，有的针对性强，有的功能全面。针对某一具体问题，在选择程序语言时，需要考虑不同语言的适用程度以及现实的可行性。如是否简单易学、语句是否容易有二义性、是否能满足对问题求解的需要、编译程序的效率等。例如，FORTRAN、C++ 等语言对数值计算有更多的优势；C 语言更广泛用于操作系统和编译器

的开发;在实时控制领域,汇编语言依然在广泛应用;而一些有特殊用途的语言,如 PHP,则专门用于网页设计,等等。

总体上,在进行程序设计时,通常从以下 3 个方面考虑程序语言的选择:

(1) **人的因素**。编程小组的人精通这门语言吗? 如果不精通,需要多长时间来学习?

(2) **语言的能力因素**。这门语言支持开发所需要的一些功能吗? 它能跨平台吗? 它有数据库的接口功能吗? 它能直接控制声卡采集声音吗?

(3) **其他因素**。这门语言开发这类任务通常的开发周期是多长? 这门语言是否被经常使用?

有的时候可能没有多少选择,比如要通过串行口控制一个外部设备,C 语言加上汇编语言是最明智的选择;而另外一些时候,选择会比较多。了解一些流行的语言,哪怕自己并不精通,对于做出合理的选择会有较大的帮助。

另外必须认识到的一点是,程序设计语言自身也在不断发展。如 2001 年微软公司推出的 C# 语言,就既拥有 C 和 C++ 语言的强大功能,又具有了 Visual Basic 易使用的特点。

2. 调试

程序编制可以在计算机上进行,也可以在纸张上进行,但最终要让计算机来运行则必须输入计算机,并经过调试,以便找出语法错误和逻辑错误,然后才能正确地运行。因此,程序调试的目的就是诊断和改正程序中可能存在的错误。

不同语言的运行环境可能差别较大,但调试纠错这一步都是必须经过的。图 1-26 所示为一段 Visual Basic 程序在 Visual Studio 2010 环境下编译器报告的语法错误。编译器在下方显示出了程序存在的错误及其位置。



图 1-26 编译环境对程序的出错报告

一般说来，语言的检查功能只能查出语法错误，即程序是否按规定的格式书写，而逻辑错误的排查则需要程序员自身的能力。比如，编写程序找两个数中的较大者，写成了如下的形式：

```
If a>b Then
    tmp=a
  Else
    tmp=b
End If
```

以上两行语句的意思是：如果 a 大于 b，将 a 赋值给 tmp，否则，a 赋值给 tmp。这就是有逻辑上的错误，因为总是找到第一个数，使程序在有些情况下得不到正确结果。不幸的是，到目前为止，编译程序还是检查不出此类错误。

1.4.4 系统测试

测试是为了发现错误而执行程序的过程。它根据整个问题求解过程中各个阶段的文档（该阶段的设计要求和目标），验证系统设计的正确性。它包括所有需求的功能是否被正确的实现、可靠性是否满足等。测试活动主要有 4 类：

（1）单元测试。对一个模块或几个模块组成的小功能单元作测试。

（2）集成测试。最终将本项目所有模块集成，交出完整的程序产品。

（3）确认测试。验证是否与需求规格说明的描述相符。包括：所有功能需求均满足；所有性能需求均达到；所有文档均已改正；其他需求已满足，等等。

（4）系统测试。对包括硬件、软件以及其他相关设备集成在一起的测试。主要测试可恢复性、安全性、抗无意或恶意攻击的强度等。

计算机软件的整个测试过程可以用图 1-27 示意。

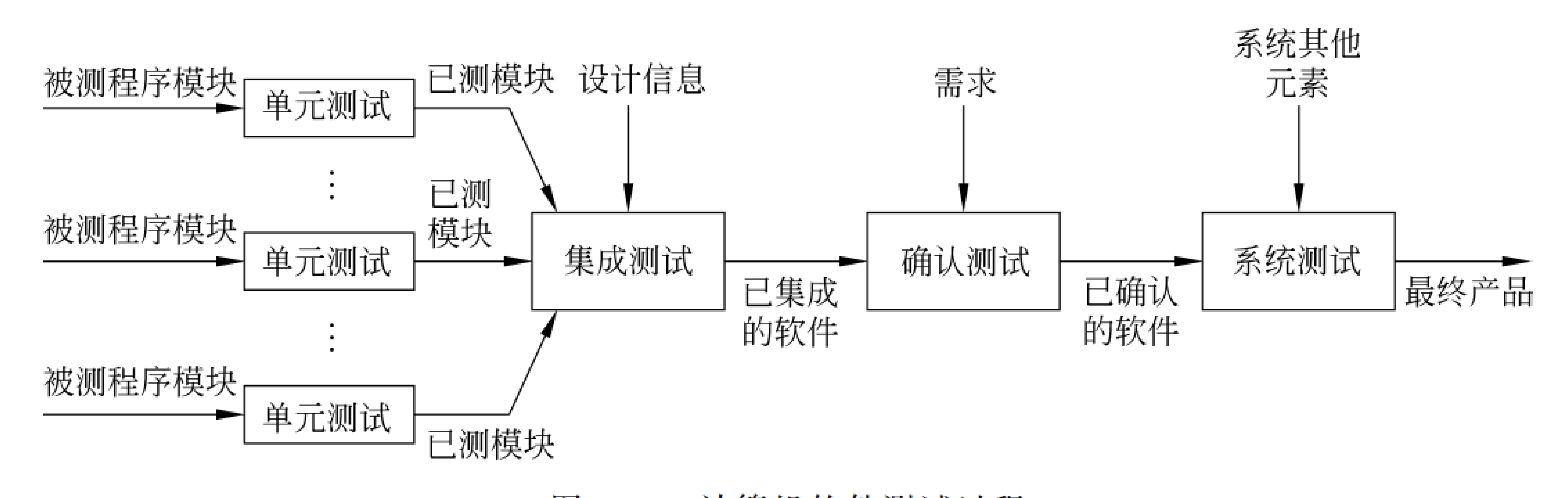


图 1-27　计算机软件测试过程

软件测试的方法有两种，一种称为白盒测试，另一种称为黑盒测试。白盒测试的对象是程序源码，主要用于单元测试。黑盒测试作为一种方法，除单元测试外，也可用于集成测试或确认测试。

黑盒测试是对功能的测试，这种方法是将软件模块看作一个黑盒子，不关心其内部的逻辑结构，而只检查在一定的输入下，其输出是否符合功能要求。

白盒测试是测试模块内部操作是否正确，即需要测试程序的内部逻辑结构。

无论是黑盒测试还是白盒测试，都不可能将所有的输入数据拿来做测试，那样所需要的时间可能是一个天文数字。举一个简单的例子。对图 1-28 所示的程序模块 P，使其在 32 位字长的计算机上运行，输入 X、Y 为整数。若将所有的 X、Y 值用做黑盒测试的输入，其最大的测试数量为 $2^{32}\times 2^{32}=2^{64}$。如果程序 P 测试一组 X、Y 数据需要 1ms，一天工作 24 小时，一年工作 365 天，则完成 2^{64} 组数据的测试需要 5 亿多年的时间。这是一个不可能完成的任务。因此，真正的测试会采用一些特定的方法。例如，对可能的输入数据进行分类，每一类选择几个代表作为测试数据，这样就可极大地减少测试的工作量和时间。

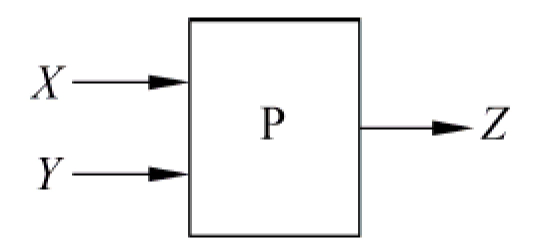


图 1-28　黑盒测试

白盒测试与之类似，设计好的测试用例既可以达到有效测试的目的，又可以提高测试的效率。有关软件测试的详细描述可参阅其他相关书籍。

在上述整个问题求解的过程中，确定需求，或者说问题的定义，是最困难的工作。对初学者来讲，由于缺乏大型系统的开发经验，甚至可能从来没有做过任何一个软件产品的设计，所以理解起来比较困难。系统测试部分也存在类似的困惑。本书介绍这些初学者难以理解的内容，目的是希望使读者能够对整个问题求解的过程有基本的概念，从而对后面的内容能够更好地体会。

*1.5　计算机科学研究前沿技术简介

作为现代科学体系的基石之一，计算机科学已逐渐成为其他学科研究的基础，并通过与其他学科的交叉研究，演变为一种横向型科学技术，从而也使计算机科学的研究领域越来越广泛。限于篇幅，本节仅简要介绍计算机科学研究的部分新技术。

1.5.1　高性能计算

计算机诞生的主要诱因就是数值计算。科学计算是计算机最初主要的甚至唯一的功能。第一台电子计算机完成一次加法运算的时间约需 0.2ms，这在当时已是了不起的高速度。但随着信息社会的发展，人类对信息处理能力的要求越来越高，不但科学研究、石油勘探、气象预报、金融保险、航空航天等需要高速的数据处理能力，甚至网络游戏都对高性能计算有了需求。现在，作为计算机科学的一个研究分支，高性能计算及高性能计算机已成为信息领域的前沿技术，在保障国家安全、推动国防科技进步、促进尖端武器发展方面具有直接推动作用，是衡量一个国家综合实力的重要标志之一。

1. 什么是高性能计算

高性能计算(high performance computing)主要研究并行①算法、并行软件技术及系

① 并行(parallelism)是指多个任务在同一时刻同时运行。

统体系结构，并致力于研制高性能计算机(high performance computer)。

从字面意思看，“高性能计算机”就是性能指标高的计算机。计算机的哪些指标要达到何种程度才能称之为“高”呢？如同计算机技术在不断发展一样，“高性能计算机”的衡量标准也在不断变化中，目前主要以计算速度(尤其是浮点运算速度)作为标准。

高性能计算的发展始于20世纪80年代，以数值模拟为主体。整个80年代，在美国国家科学基金会、美国航空航天局及美国政府的支持下，高性能计算机的研制得到快速发展，促进了峰值速度可达10亿次每秒的Cray超级向量计算机和多CPU并行计算机等高性能计算机的问世。从20世纪90年代起，美国先后推出了“高性能计算和通信”计划和“先进模拟和计算”计划。重点是高性能计算机系统、并行算法与先进软件技术、基础研究和教育网络，以及以科学计算为基础的核武器库存可靠性和有效性研究等。

我国高性能计算机的研制历经几十年努力，也得到很大发展。我国是继美国和日本之后第三个具备高性能计算机系统研制能力的国家。如国防科技大学2010年研制的“天河一号A”的实测运算能力达2507万亿次每秒，而在2013年研制出的“天河二号”，其峰值计算速度更是达到5.49亿亿次每秒。如今，高性能计算在核武器模拟、气候变化分析和模拟、量子模拟等重要科技领域已得到有效应用，并取得了十分重要的成果。

2. 高性能计算机的关键技术

高性能计算机的研究涉及软硬件技术、通信技术、纳米技术等多个学科，近年的研究主要集中于大规模并行处理体系结构、高性能算法、可重构计算、功耗等方面。

大规模集成电路技术对计算机性能的提高无疑有很大的影响，而体系结构的优劣同样是影响计算性能的重要因素。先看一些生活中随处可见的案例：

① 节日的高速公路上，因某辆车出现故障而造成的数千米车辆拥堵。

② 干旱时排队接水的人形成长龙，水却流得很慢。

③ 虽然人很多，但工作任务却只分给了少数几个人。结果忙的忙“死”，闲的闲“死”。

……

生活中的这些问题反映到计算机系统中，就是体系结构问题。由于存储器的存取速度和处理器的计算能力之间存在较大的不平衡，使处理器可以进行高速计算，但计算需要的数据存储器却来不及提供(如上例中的①和②)，即数据访问与计算能力存在较大差距[①]。

实现高性能计算的手段之一是并行处理多个任务。这里的并行是指将计算任务分配到系统中的各个计算资源上，使之同时工作，以使整体计算能力得到充分发挥。能否保证每个计算部件都既无空闲或等待，也不存在过负载(如上例中的③)，这是高性能计算中又一个很重要的研究课题——负载均衡。负载不均衡将会导致计算效率的大幅下降。

① 这一问题称为存储器墙。

每秒千万亿次的计算机系统需要并行度和并行效率更高的算法。随着所求解问题规模的增大，需要新的物理模型，反过来又需要不同的求解方法。与软硬件性能提高相比，算法改进对应用问题求解的性能影响会更加显著，因此面向应用问题进行的高性能算法设计也是超级计算机应用的重要基础。除高效算法外，高性能计算机在软件技术的研究方面还包括操作系统、面向新一代体系结构的编译优化和实现技术等。

目前的高性能计算机的主流体系结构都是基于冯·诺依曼的计算机理论基础。传统计算机体系结构存在如存储器墙、I/O 通信、功耗、程序复杂性等问题。随着技术的发展，现代计算机系统规模不断增大，系统结构越来越复杂，创新高性能计算体系结构已是大势所趋。

1.5.2 普适计算

只有当机器进入人们生活环境而不是强迫人们进入机器世界时，机器的使用才能像林中漫步一样新鲜有趣。

案例 1：在一个智能教室环境下，如果投影设备的显示效果不是很理想，教师可以通过自己的掌上电脑向学生的掌上电脑发送电子课件。当教师走近学生讨论组时，其掌上电脑会动态加入该组，下载该组正在讨论的材料。

这就是一个普适环境，它由投影机、教师掌上电脑和学生掌上电脑组成，该系统通过可重新配置的上下文敏感中间件，突出对环境的感知和动态自组网络通信的支持。

案例 2：一个普适医疗服务系统可以提供任何时间、任何地点的医疗服务访问。在一辆急救车上配备无线定位系统，就可准确定位突发事故现场，同时利用无线网络获取实时的交通信息。另外，在事故现场，通过便携式或移动式设备监测病人的脉搏、血压、呼吸等数据，通过无线网络访问分布式的医疗服务系统，下载有关病历数据等必要信息。

除了基于定位系统的应急响应机制，普适医疗服务系统的功能还包括基于移动设备和无线网络的远程医疗诊断、远程病人监护，以及远程访问具有患者病历信息的医疗数据库。

施乐公司 Palo Alto 研究中心的首席技术官 Mark Weiser 曾经说过，最深刻和强大的技术应该是“看不见”的技术，是那些融入日常生活并消失在日常生活中的技术。这个被称为“普适计算之父”的人在 20 世纪 90 年代初就声称：受社会学家、哲学家和人类学家的影响，他重新审视了网络计算模式。他指出，21 世纪的计算将是一种无所不在的计算(ubiquitous computing)。

因此，普适计算也称为普及计算(pervasive computing 或者 ubiquitous computing)，强调将计算和环境融为一体，而让计算本身从人们的视线中消失，使人的注意力回归到要完成的任务本身。它的含义是：在普适计算的模式下，人们能够在任何时间、任何地点以任何方式进行信息的获取与处理。或简单地说，是一种无处不在的计算模式。

互联网应用的兴起使计算模式继主机计算和桌面计算之后进入一种全新的模式，也就是普适计算模式。这种新的计算模式强调把计算机嵌入到人们日常生活和工作环境

中，使用户能方便地访问信息和得到计算服务。

普适计算当然包括移动计算，但普适计算更强调环境驱动性。这要求普适计算对环境信息具有高度的可感知性，人机交互更自然化，设备和网络的自动配置和自适应能力更强，所以普适计算的研究涵盖传感器、人机交互、中间件、移动计算、嵌入式技术、网络技术等领域。

1.5.3 云计算

云计算(cloud computing)概念是由 Google 公司提出的，是分布式计算、并行计算和网格计算的发展，或者说是这些科学概念的商业实现，指通过网络以按需、易扩展的方式获得所需的服务。

云计算的核心思想是将大量用网络连接的计算资源统一管理和调度，构成一个计算资源池向用户按需服务。提供资源的网络被称为“云”。“云”中的资源在使用者看来是可以无限扩展的，并且可以随时获取，按需使用，随时扩展，按使用付费。这就像用水、电一样付费使用。

云计算的基本原理是通过使计算分布到大量的分布式计算机上(而非本地计算机或远程服务器中)，使得企业能够将资源切换到需要的应用上，根据需求访问计算机和存储系统。

在云计算模式下，用户不再需要购买复杂的硬件和软件，而只需要支付相应的费用给云计算服务提供商，通过网络就可以方便地获取所需要的计算、存储等资源。从服务的角度，云计算是一种全新的网络服务模式，将传统的以桌面为核心的任务处理转变为以网络为核心的任务处理，利用互联网实现自己想完成的一切处理任务，使网络成为传递服务、计算力和信息的综合媒介，真正实现按需计算、网络协作。

1.5.4 人工智能

人工智能(artificial intelligence)是研究、开发用于模拟、延伸和扩展人的智能的理论、方法、技术及应用系统的一门新的技术科学。它是计算机科学的一个分支，它企图了解智能的实质，并生产出一种新的能以人类智能相似的方式做出反应的智能机器。

人工智能的基本研究内容主要包括以下几个方面：

(1) 机器感知。主要包括计算机视觉和计算机听觉，研究用计算机来模拟人和生物的感官系统功能，使计算机具有“感知”周围世界的能力；具体来说，就是让计算机具有对周围世界的空间物体进行传感、抽象、判断的能力，从而达到识别、理解的目的。根据其处理过程的先后及复杂程度，计算机视觉的任务可以分成下列几个方面：图像的获取、特征抽取、识别与分类、三维信息理解、景物描述和图像解释。计算机听觉建立在机器识别语言、声响和自然语言理解的基础上。语言理解包括语音分析、词法、句法和语义分析。机器感知是计算机获取外部信息的基本途径，是使机器具有智能不可缺少的组成部分，对此人工智能中已经形成两个专门的研究领域：模式识别和自然语言理解。

(2) 机器思维。指计算机对通过感知得来的外部信息及及其内部的各种工作信息进

行有目的的处理。正像人的智能来源于大脑的思维活动一样，机器智能也是通过机器思维实现的，因此，机器思维是人工智能研究中最重要、最关键的部分。为了使计算机能模拟人类的思维活动，需要开展以下几个方面的研究：

- 知识的表示，特别是各种不精确、不完全、非规范知识的表示。
- 知识的组织、累积和管理技术。
- 知识的推理，特别是各种不精确推理、归纳推理、非单调推理、定性推理。
- 各种启发式搜索及控制策略。
- 神经网络、人脑的结构及其工作原理。

(3) 机器学习。学习是人类具有的一种重要智能行为，人类能够获取新知识，学习新技巧，并在实践中不断完善、改进。机器学习就是要使计算机具备这种学习能力，在不断重复的工作中对本身能力的增强或者改进，使得在下一次执行同样任务或类似任务时，会比现在做得更好或效率更高，并且能克服人类在学习中的局限性，如遗忘、效率低、注意力分散等。

(4) 机器行为。与人的行为能力相对应，机器行为主要是指计算机的表达能力，如“说”、“写”、“画”等。对于智能机器人，它还应具有人的四肢功能，能走路，能操作。

(5) 智能系统及智能计算机构造技术。人工智能的最终目标就是要构造智能系统及智能机器，因此需要开展对系统分析与建模、构造技术、建造工具及语言的研究。

1950 年，计算机理论的奠基人艾伦·图灵在哲学性杂志《精神》上发表了一篇题为《计算机和智能》(*Computing Machinery and Intelligence*)的著名文章，文章提出了一个检验计算机是否具备人类“思维”的方法，后来被称为“图灵测试”或“图灵检验”。

被测试者一个是人，另一个是声称有人类智力的机器。测试时，测试人与被测试者分开，测试人通过一些装置(如键盘)向被测试者提出问题，这些问题可以是任何问题。提问后，如果测试人能够正确地分出两个被测试者谁是人谁是机器，那么机器就没有通过图灵测试；如果测试人没有分出这两者，则这个机器就是有人类智能的。

当然，目前还没有一台计算机能够通过图灵测试，也就是说，计算机的智力与人类的智力还相差很远。但图灵指出：“如果机器在某些现实的条件下，能够非常好地模仿人回答问题，以至提问者在相当长时间里误认为它不是机器，那么机器就可以被认为是能够思维的。”

虽然成功通过图灵测试的计算机还没有出现，但已有计算机在测试中“骗”过了测试者。著名的“深蓝”(DeepBlue)机器人就是一个很好的例证。1997 年 5 月 11 日，由 IBM 公司研制的、起名为“深蓝”的超级计算机 AS/6000 SP，与“人类最伟大的棋手”、苏联国际象棋世界冠军卡斯帕罗夫进行的人机象棋大赛，最终计算机以微弱优势取胜。

这个案例以及众多的科幻影视作品都不禁会让人设想：未来会出现能够骗过大多数人的计算机吗？

1.5.5 物联网

物联网(the internet of things)，顾名思义，就是“物物相连的互联网”，是新一代信息

技术的重要组成部分。它是通过射频识别(Radio Frequency Identification,RFID)、红外感应器、全球定位系统、激光扫描器等信息传感设备,按约定的协议,把任何物体与互联网相连接,进行信息交换和通信,以实现对物体的智能化识别、定位、跟踪、监控和管理的一种网络。

物联网的核心和基础仍然是互联网,是在互联网基础上延伸和扩展的网络。其用户端可延伸和扩展到任何物体与物体之间,实现物体与物体之间的信息交换和通信。

物联网可分为3层:感知层、网络层和应用层(如图1-29所示)。

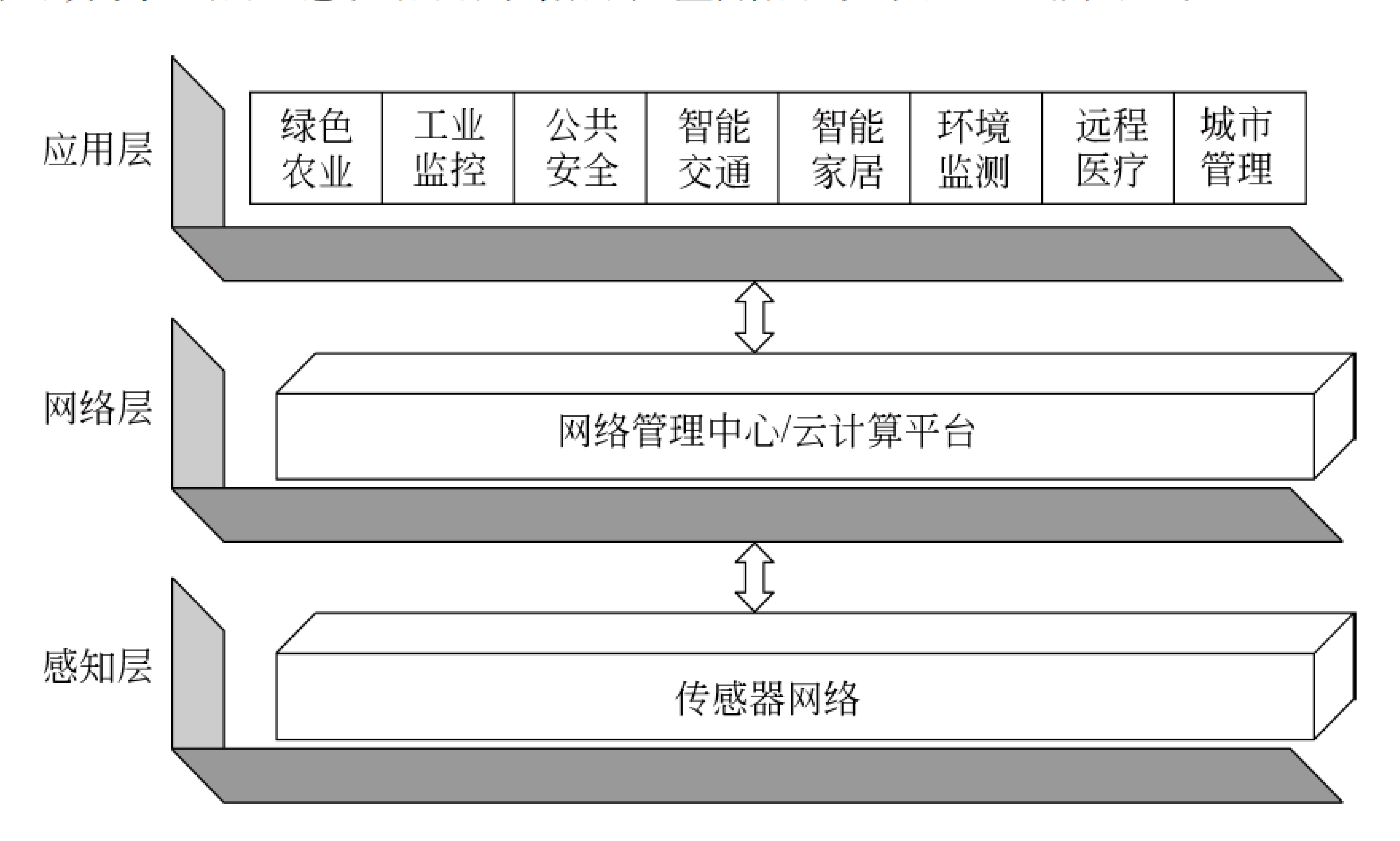


图1-29　物联网架构示意图

感知层由各种传感器(如温度传感器、湿度传感器、摄像头、GPS等感知终端)和传感器网关构成。其作用相当于人的眼耳鼻喉和皮肤等神经末梢,它是物联网识别物体、采集信息的来源,其主要功能是识别物体和采集信息。

网络层由各种私有网络、互联网、有线和无线通信网、网络管理系统和云计算平台等组成,相当于人的神经中枢和大脑,负责传递和处理感知层获取的信息。

应用层是物联网和用户(包括人、组织和其他系统)的接口,它与行业需求结合,实现物联网的智能应用。

目前,物联网技术已在多个行业领域得到应用。例如,上海浦东国际机场的入侵防护系统,为了保护机场安全,铺设了3万多个传感结点,覆盖了地面、栅栏和低空探测,可以防止人员翻越、偷渡、恐怖袭击等攻击性入侵。

物联网技术近年来发展迅速,已广泛应用于物流、零售、制药、安保等各个领域,在不断地改变着人们的生活方式。未来的物联网将会向更加智能化的方向发展。

习　题

一、填空题

1. 一个计算机系统由(　　)系统和(　　)系统两部分组成。

2. 说明以下计算机中的部件是属于主机系统、软件系统、还是属于外部设备。

(1) CPU （ ）

(2) 内存条 （ ）

(3) 网卡 （ ）

(4) 键盘和鼠标 （ ）

(5) 显示器 （ ）

(6) Windows 操作系统 （ ）

3. 外部设备与主机要进行信息交换，必须要通过（ ）。

4. 控制芯片组是主板的核心部件，它由（ ）部分和（ ）部分组成。

5. 软件系统包括（ ）软件和（ ）软件。

6. 图灵机模型主要由（ ）、（ ）、（ ）和（ ）4 个部分组成。

7. 能够被计算机解决的问题的特点是（ ）。

8. 在模型建立的前提下，利用计算机求解问题的核心工作就是（ ）设计。

9. 要使一个问题能够用计算机解决，其必要条件是（ ）。

10. 第一代计算机的主要部件是由（ ）构成的。

11. 未来全新的计算机技术主要指（ ）、（ ）和（ ）。

12. 未来电子计算机的发展方向是（ ）、（ ）、（ ）和（ ）。

13. 软件的测试方法包括（ ）和（ ）。

14. 普适计算的主要特点是（ ）。

二、简答题

1. 图灵机模型主要由哪 4 个部分组成？
2. 图灵机在形式上可以用哪 5 个元素描述？它们分别表示什么含义？
3. 图灵机模型中的 4 个要素是什么？
4. 简述图灵机的工作过程。
5. 简述什么是计算。
6. 简述问题求解的一般过程。
7. 简述高性能计算机涉及的主要关键技术。



章 信息的表示编码

引言

今天,计算机已由最初仅能实现数值计算的计算工具发展为能够存储和处理各种信息的综合处理系统。它不仅能做各种复杂的数值计算,还能处理各种音频、视频信息等。由于数字计算机由各种逻辑器件构成,而逻辑器件只有“真”和“假”、“高”和“低”、“通”和“断”等这样两种状态。因此,所有为计算机存储和处理的信息都必须转换为只用两种状态表示,即二进制编码形式。本章首先讨论计算机为什么会选择二进制,然后介绍各种信息在计算机中的表示和编码,以及后续编写程序时将会涉及的计算机中的数制、二进制数的表示和运算等。

教学目的

- 理解计算机为什么采用二进制。
- 理解计算机中的信息表示与编码方法。
- 理解计算机中常用记数制的表示及其相互间的转换。
- 了解二进制数的表示。
- 理解机器数的表示及运算。

2.1 计算机与二进制

或许是由于人类有 10 个手指的缘故,从结绳记数时代开始,人类就习惯于用十进制计数。在十进制计算系统中,每一个十进制数字都是一个书写符号。也就是说,要表示一个十进制数,至少需要 10 个符号。早期的机械式计算装置就采用十进制。它利用齿轮的不同位置来表示不同的数值。比如将 10 个不同大小的齿轮级联在一起,每个齿轮设为 10 个齿(或称 10 个格),分别表示 0~9。小齿轮每转一圈,比它大一级的齿轮走 1 格(就像钟表里的秒针每走 1 圈,分针就走 1 格一样)。这样,就可以表示 0~9 999 999 999 范围的数据了。

诞生于1946年的第一台电子计算机ENIAC采用的也是十进制，可以同时处理10个十进制数。但是，由于十进制有10个符号，意味着需要有10种稳定状态与之对应，不仅造成数据量大、工作速度低，更主要是用电子器件实现起来很困难。所以，十进制计算机没有能够得到推广。

香农在他的《通信的数学理论》论文中曾首次指出，通信的基本信息单元是符号(symbol)，而最基本的符号是二值符号。二值符号的一个例子是：在一根预定的导线上出现一个电脉冲，于是脉冲的存在或不存在(即符号的"值")就传送了信息；另外，开关的断开或闭合、脉冲的正极性或负极性等，都属于二值符号体系。

所以，这样看来，两种状态(二值)的表示是很容易实现的。例如，一组两根导线的系统，就是两个二值符号的集合。如果将电脉冲的出现用1表示，不出现用0表示，则这个两根导线的系统就可以表示4种状态：

电脉冲都不出现——00；仅一根上出现，另一根不出现——01或10；都出现——11。

同样道理，一组3根传输电脉冲的导线系统就有8种状态；一组4根导线系统就有16种状态，依此类推，导线越多，能够表示的状态数就越多(要表示10个符号无非用一组4根导线的系统就足够了)。而这一切的基本点，就是导线上电脉冲的出现和不出现(实现这一点可是非常容易的)。回归到符号系统，就是0和1，这也就是今天计算机中普遍采用的二进制。

1701年，德国数学家莱布尼茨(G. W. Leibniz)发明了二进制①。莱布尼茨的二进制就是用0和1表示一切数字，如000、001、010、011就分别代表0～3这4个数字。1848年，英国数学家乔治·布尔(George Boole)推出了二进制运算法则，为二进制计算机的诞生奠定了基础。

现代计算机采用二进制计数。采用二进制的理由主要有以下几点：

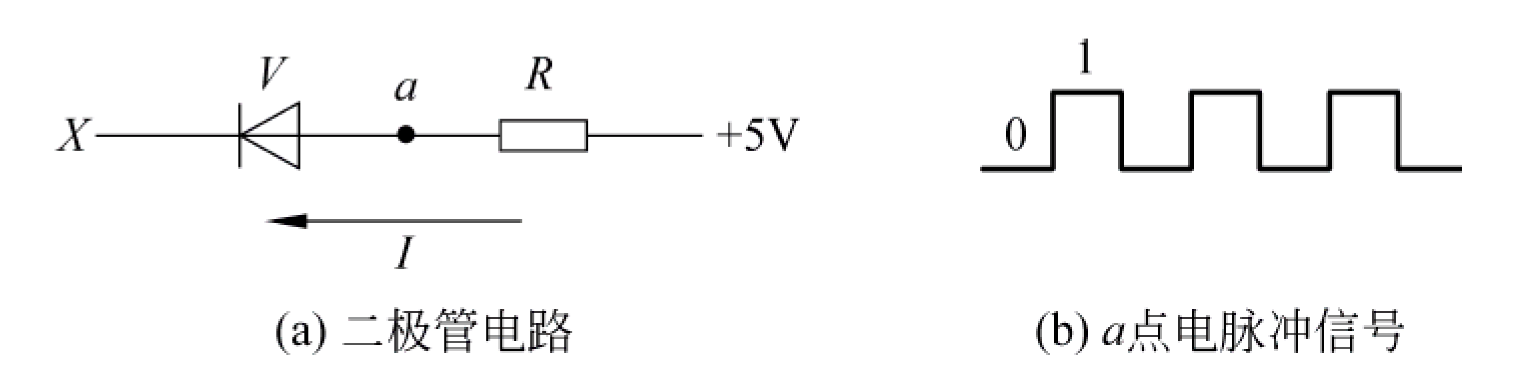
(1) 技术实现简单。二进制只有0和1两个基本符号，任何两种对立的物理状态都可以归结为二进制表示。例如，开关的"闭合"与"断开"，电位的"高"和"低"，晶体管的"导通"与"截止"，电容的"满电荷"与"空电荷"，等等。如此，一切有两种对立稳定状态的器件都可以表示二进制的0和1。

图2-1中，当(a)图中的X端电位为0V时，晶体二极管导通，有电流流过电阻R；当X端电位为+5V时二极管将截止，R上将不会有电流流过。根据欧姆定律知，导通时a点电位≈0V(低电平)；截止时因电流$I=0$，则a点电位=+5V(高电平)。如果周期性地使X端呈现0V和+5V(如图2-1(b)所示的脉冲波)，则二极管就会周期性地导通和截止。如果将0V用0表示，5V用1表示，则上述过程就与二进制码对应了。

具有两种稳定状态的电子元件很容易找到，产生两种稳定状态的电路也易于设计。因此，计算机采用二进制的重要原因之一就是其非常容易用电子器件实现，可靠性也高。

(2) 运算规则简单。二进制的算术运算特别简单，加法和乘法各仅有3条运算规则(加法：0+0=0，0+1=1，1+1=10；乘法：0×0=0，0×1=0，1×1=1)。二进制减法和

① 胡阳、李长铎在《莱布尼茨——二进制与伏羲八卦图考》一书中论证了莱布尼茨的二进制至少在某种程度上受到了中国伏羲八卦图的启发。



(a) 二极管电路　　　　(b) a点电脉冲信号

图 2-1　二极管的导通和截止

除法则可以通过一定的变换转换为加法和乘法运算①。

(3) 易于与十进制之间的数值转换。从符号体系的角度来讲，二进制是二值符号体系，十进制需要 10 个符号，4 个具有两种状态的二值符号就可以组合出 10 个符号。就像上文中讲到的 4 根传输电脉冲的导线，每根导线上脉冲的有和无就表示两种状态，而 4 根导线上脉冲的有和无就可以组合出 10 种状态了（当然，最多可以组合出 16 种状态）。所以，虽然人类不习惯二进制记数，但将二进制转换为十进制是很容易实现的。

(4) 适合逻辑运算。逻辑运算②的对象是“真”和“假”，二进制数的 1 和 0 正好可与逻辑值“真”和“假”相对应，这就使计算机进行逻辑运算变得非常方便。比如图 2-1(a)所示的电路中，当 X 端为低电位时，a 点输出低电位；若 X 端为高电位，则 a 点输出高电位。如果将低电位设为“假”，用 0 表示，高电位设为“真”，用 1 表示，则电位的“高”和“低”就与二进制的 1 和 0 以及逻辑的“真”和“假”形成了对应的关系。

为了对逻辑运算先有一个直观的感觉，再来看一个二极管电路的示例。

在图 2-2 所示的电路中，若二极管 V_1 或 V_2 导通，说明 X_1 或 X_2 一定是低电位（可以假设是 0V）。如果将 X_1、X_2 作为这个二极管电路的输入，Y 作为输出，将低电位用“假”表示，高电位用“真”表示，则由图 2-2 就可以得出表 2-1 所示的关系。

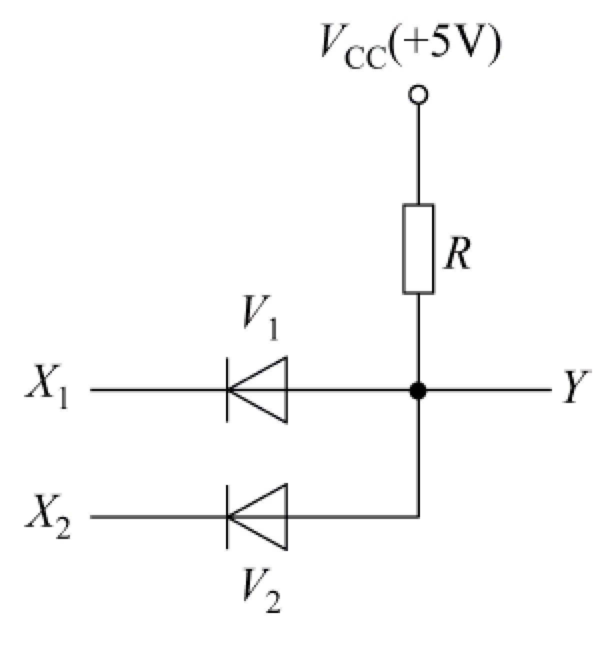


图 2-2　二极管逻辑电路

表 2-1　示例电路的输入与输出

输　入		输　出
X_1	X_2	Y
假	假	假
假	真	假
真	假	假
真	真	真

即，输入 X_1 和 X_2 有任意一个为假，则输出 Y 为假；只有当 X_1、X_2 均为真时，Y 为真。表 2-1 所示这个关系称为逻辑“与”。

计算机可以说就是由许许多多个类似图 2-2 所示的逻辑电路组成的，在这样的电路中，高电位用 1 表示，低电位用 0 表示。类似的还可以是：电路的导通用 0 表示，截止用 1 表示；开关的闭合用 0 表示，断开用 1 表示，等等。如此，二值符号体系就与实际的电路实

---

① 将减法运算转换为加法运算的原理请参见 2.4 节（二进制数的表示和运算），除法到乘法的转换请参阅有关计算机原理方面的书籍。

② 关于逻辑运算的详细介绍见 3.1 节（逻辑代数基础）。

现联系在了一起。

以上就是计算机为什么没有选择人类最习惯的十进制，而选择了二进制的主要理由。今天，无论计算机的功能有多么强大，它能够处理的信息有多么丰富，若除去各种辅助的软件，计算机硬件唯一能够直接识别的信息只有一种，就是 0 和 1。只是这里的 0 和 1 不只是数学上的数字概念，还反映着诸多的物理属性。如事情的“好”和“坏”、东西的“有”和“无”、电平的“高”和“低”、事实的“真”和“假”等。

2.2 计算机中的信息表示与编码

如今，信息是一个非常流行的词汇。人际社会中，每天都少不了信息交互，每个人都是信息的发布者，同时也都是信息的接收者。互联网上，更是每分每秒都有大量的信息在传送。信息交换和信息共享促进了新知识的传播、新价值的产生，也推动着社会的进步。

在这个“信息爆炸”的时代，对信息的传播、处理和存储都离不开计算机这个载体。本节就在给出“信息”一词一般描述的基础上，介绍计算机中的信息表示方法。

2.2.1 什么是信息

对“信息”一词最早的解释见于哈特莱（Ralph V. L. Hartley）1928 年发表在《贝尔系统技术杂志》上的《信息传输》一文中，他把信息理解为选择通信符号的方式，并用选择的自由度来计量这种信息量的大小。信息论（Information Theory）的开山鼻祖、美国数学家香农（C. E. Shannon）1948 年在《贝尔系统技术杂志》上发表了一篇题为《通信的数学理论》的论文，该文被认为是信息论诞生的标志。香农以概率论为工具，阐述了通信工程中的一系列基本理论问题，建立了信息从信源（发送方）通过信道（传输途径）传递给信宿（接收方）的通信系统模型，并给出了计算信源信息量和信道容量的方法和计算信息熵的公式。他对信息的解释是：信息是用来减少随机不定性的东西。控制论创始人之一，美国科学家维纳（N. Wiener）指出：信息就是信息，既不是物质也不是能量。他专门指出了信息是区别于物质与能量的第三类资源。

《辞源》中将信息定义为“信息就是收信者事先所不知道的报道”。作为科学术语，可以简单地将信息理解为“消息接收者预先不知道的报道”。我国学者钟义信从认识论的层次将信息定义为“主体关于某事物的认识论层次信息，是指主体所感知或表述的关于该事物的运动状态及其变化方式，包括状态及其变化方式的形式、含义和效用”。

对于信息的定义，至今仍是众说纷纭，莫衷一是。但人们对信息的共同认识是：信息是一种宝贵的资源，信息、材料（物质）、能源（能量）是组成社会物质文明的三大要素。

相对于通信范围内的信息论（狭义信息论），广义信息论以各种系统、各门科学中的信息为对象，以信息过程的运动规律作为主要研究内容，广泛研究信息的本质和特点，以及信息的取得、计量、传输、储存、处理、控制和利用的一般规律，使得人类对信息现象的认识与揭示不断丰富和完善。所以，广义信息论也被称为信息科学，它以信息为主要研究对

象，是一门新兴的跨多个学科的科学。

在一般用语中，信息、数据、信号并不被严格区别，但从信息科学的角度看，它们是不能等同的。在用现代科技（计算机技术、电子技术等）采集、处理信息时，必须将现实生活中的各类信息转换成智能机器能识别的符号（符号具体化即是数据，或者说信息的符号化就是数据），再加工处理成新的信息。数据既可以是传统概念上的数值，也可以是文字、声音或图像等，是信息的具体表示形式，是信息的载体。而信号则是数据的电磁或光脉冲编码，是各种实际通信系统中适合信道传输的物理量。信号可以分为模拟信号（随时间而连续变化的信号）和数字信号（在时间上的一种离散信号）。

计算机在诞生之初所存储和处理的信息只有数值信息。但随着技术的发展，各种非数值信息也需要计算机处理。所以，现代计算机所存储和处理的信息还包括文字、声音、图像等各种非数值信息。

2.2.2 数值信息表示

“计算机”（computer）一词，顾名思义，是用来计算的，也就是说计算机是一种计算装置。的确，研制计算机最初的目的就是计算①。所以，早期计算机所处理的信息都是数值信息。从 2.1 节的讨论中，我们已经知道，计算机唯一能够直接识别的数值只有 0 和 1 这样的二进制码。那么，如何将现实世界中的那些需要计算机处理的各种非数值信息与 0 和 1 联系起来呢？方法就是所谓的数字化。

数字化（digitizing，digitization）简单地讲，就是将各种信息转换为能用 0 和 1 表示的过程。一位 0 或一位 1 就是计算机中信息的最小单位，称为 1 个比特（1b）。

1. 比特

比特也称为位（b）。它表示逻辑器件的一种状态：“断开”或“闭合”。在计算机的内存储器中，它可以是用于存放信息的晶体管的“开”或“关”，也可以是某个电容的充电或放电；在硬磁盘中，位通过磁盘盘片表面的磁场方向表示（“南—北”或“东—西”）；在常用的 CD-ROM 光盘上，它是光的反射与否；而在计算机所处理和存储的数字音频信号中，可以用 1 表示高音，用 0 表示低音。

一串比特可以代表一个数据。例如，1000、0100、0010、0001，这一串比特可以代表十进制的 8、4、2、1。另外，一串比特也可以代表一组文字（字符）。例如，01000001、01000010、01000011，这一串比特就表示 A、B、C。当然，由于计算机只认识 0 和 1，所以，像声音、图像等各种需要计算机处理的信息都需要用 0 和 1 表示，即一串比特也可以可能是一段音乐或一段视频图像的代表符号。

一个十进制数可以由多个数位构成。例如，128 就是由 3 个数位构成，最右边是个位，也是这个十进制整数的最低位，其权值是 10^0。之后，从右向左，依次是十位（次低位）和百位（最高位），相应的权值分别是 10^1 和 10^2。

---

① 研制世界上第一台通用数字计算机 ENIAC 的主要目的就是为了更精确地计算弹道轨迹和火力表。

同样，二进制数的位因其处于数的不同位置也具有不同的权值，其权值的大小也是从右向左依次增加。如二进制数 1011，同样最右边是最低位，权值为 2^0；之后从右向左，其权值分别为 2^1、2^2，最高位的权值为 2^3。由于最低位的权值是 2^0，因此在计算机中，常用 bit0 来表示一个二进制数的最低位，高位则依次为 bit1，bit2，…。

对一个二进制数，哪一位是最低位，哪一位是最高位，且最低位称为“第 0 位”（**不是“第 1 位”**），这些是初学者必须要清楚的问题。

2. 字节

一位 0 或 1 无法表示太多数据，需要将多位组合起来。由于计算机对数据的处理多以 8 位二进制码（或 8 位的整数倍）为单位，所以常将 8 位二进制码作为一个整体，称为 1 字节（1B）。1B 是 8 位二进制码，能够表示的最大数是 $2^8-1=255$。

字节是计算机中表示存储空间大小的基本容量单位。例如，计算机内存的存储容量、磁盘的存储容量等都是以字节为单位表示。此外，为表示更大的数字，将更多字节结合起来，如 2 字节是 16 位，能够表示的最大数就是 $2^{16}-1=65\ 535$。依此类推，就有了以下这些表示大数据的单位：千字节（KB）、兆字节（MB）、吉字节（GB）、太字节（TB）等。它们之间的换算关系如下：

$$1\text{B}=8\text{b}$$
$$1\text{KB}=2^{10}\text{B}=1024\text{B}$$
$$1\text{MB}=2^{10}\text{KB}=2^{20}\text{B}=1024\text{KB}$$
$$1\text{GB}=2^{10}\text{MB}=2^{20}\text{KB}=2^{30}\text{B}=1024\text{MB}$$
$$1\text{TB}=2^{10}\text{GB}=2^{20}\text{MB}=2^{30}\text{KB}=2^{40}\text{B}=1024\text{GB}$$

3. 字长

字长指计算机能够同时处理（专业的说法是并行处理）的二进制位数。在计算机诞生初期，受各种因素限制，计算机一次能够并行处理 8b 二进制码，即一次能够进行 8 位二进制码的加减乘除等运算。随着电子技术的发展，计算机的并行能力越来越强，从 8 位、16 位、32 位，直至今天，微型机的并行处理能力一般为 64 位，大型机已达 128 位。计算机一次能够并行处理的二进制位数称为该计算机的字长，也称为计算机的一个“字”。因此，早期的计算机被称为 8 位机，而今天的个人计算机（PC）则称为 64 位机。

字长是计算机的一个重要性能指标，直接反映了一台计算机的计算能力和精度。字长越长，计算机处理数据的速度就越快。这可以通过一个例子来说明。如计算 5×8，我们可以立即得出答案为 40。但如果要计算 55×88，就不可能立即得到正确的答案。这是因为 55×88 的运算已超出了人脑的“字长”。为了得出结果，需要将复杂的问题（如 55×88）进行分解，如分解为 50×80＋50×8＋5×80＋5×8。这样，虽然较容易得出结果，但可以看出，需要花费比较多的时间。随着数字的增大，需要花的时间就更长。

人脑是这样，计算机同样是这样。计算机能够一次直接处理的最大数决定于计算机的字长。如果要计算的数据超出了计算机的字长，就必须对数据进行分解。一台字长为 16 位的计算机，可以直接处理 2^{16}（65 536）之内的数据，对于超过 65 536 的数就必须分解

之后才能处理。32 位机比 16 位机优越的原因就在于它在一次操作中能处理的数更大(2^{32},达 40 亿)。能处理的数字越大,则操作的次数就越少,系统的效率也就越高。

2.2.3 文字信息表示

由上述分析已知,计算机能够直接识别的只有二进制码。所以,要让计算机保存或处理的所有信息都必须采用二进制码表示,文字信息也不例外。所以,不论是西文字符还是中文字符或其他国家的文字符,要使计算机能够处理,都必须转换为二进制表示。这种将信息用二进制 0 和 1 来表示的过程称为编码。

1. 西文字符的编码

文字由字符组成。计算机是美国人发明的,所以,计算机中的文字首先是西文字符,包括字母、数字、符号及特殊控制字符。西文字符编码方式很多,目前国际上广泛使用的是为美国英语通信所设计的 ASCII 码(American Standard Code for Information Interchange,美国标准信息交换码),分为标准 ASCII 码和扩展 ASCII 码两种。

标准 ASCII 码用 7 位二进制码(bit_6～bit_0)表示,总共可表示 128 个字符(知道为什么是 128 个吗?),包含英文大小写字母、数字 0～9、标点符号、非打印字符(换行符、制表符等 4 个)以及控制字符(退格、响铃等)。一个字符对应一个编码(详见附录 B)。

当然,由于计算机从诞生那天起能够并行处理的二进制数就是 8 位(从来没有 7 位)。因此,标准 ASCII 码实际上是用 8 位二进制码来表示的,8 位二进制码也称为 1 字节(Byte),在内存中占用 1 个单元("单元"的概念在第 1 章中就介绍过了),最高位(bit_7)在默认情况下设为 0。表 2-2 给出了部分字符的标准 ASCII 码。

这样,一串比特位就可以与某个字符对应起来了。如,01000010 就代表了大写英文字母 B,而 01100001 就代表小写的字母 a。存入计算机中的所有信息,不论是论文还是音乐或照片,在计算机的存储设备上都是用这样一串串的比特位来表示的。

这种将信息用一串比特位来表示的方法就叫作信息的编码。不论哪种信息,要想为计算机所处理,都必须要进行这样的编码。

思考 请对照表 2-2,尝试将"Hello World!"转换为对应的 ASCII 码。

在计算机的信息传输中,为尽量减少和避免错误,除提高软硬件系统的可靠性外,也常在数据的编码上想办法,即采用带有一定特征的编码方法。数据校验码就是这样一种能发现错误并具有自动改错能力的编码方法。

在 ASCII 码的传送中,最常用到的校验码是一种开销小、能发现一位数据出错的奇偶校验码。带有奇偶校验的 ASCII 码将最高位(bit_7)用作奇偶校验位,以校验数据传送中是否有一位出现错误。

所谓奇偶校验,是指在代码传送过程中用来检验是否出现错误的一种方法,分奇校验和偶校验两种。奇校验规定:正确的代码一个字节中 1 的个数必须是奇数,若非奇数,则

使最高位 bit_7 为 1(补为奇数);偶校验规定:正确的代码一个字节中 1 的个数必须是偶数,若非偶数,则使最高位 bit_7 为 1。

表 2-2 部分字符的标准 ASCII 码

ASCII 码	字符	ASCII 码	字符	ASCII 码	字符	ASCII 码	字符
00000000	空格	01000011	C	01010100	T	01101011	k
00110000	0	01000100	D	01010101	U	01101100	l
00110001	1	01000101	E	01010110	V	01101101	m
00110010	2	01000110	F	01010111	W	01101110	n
00110011	3	01000111	G	01011000	X	01101111	o
00110100	4	01001000	H	01011001	Y	01110000	p
00110101	5	01001001	I	01011010	Z	01110001	q
00110110	6	01001010	J	01100001	a	01110010	r
00110111	7	01001011	K	01100010	b	01110011	s
00111000	8	01001100	L	01100011	c	01110100	t
00111001	9	01001101	M	01100100	d	01110101	u
00111100	<	01001110	N	01100101	e	01110110	v
00111101	=	01001111	O	01100110	f	01110111	w
00111110	>	01010000	P	01100111	g	01111000	x
01000000	@	01010001	Q	01101000	h	01111001	y
01000001	A	01010010	R	01101001	i	01111010	z
01000010	B	01010011	S	01101010	j	00100001	!

例如,大写字母 A 的标准 7 位 ASCII 码为 1000001,具有偶校验的 A 的 ASCII 码是 01000001,而具有奇校验的 A 的 ASCII 码是 11000001。两组编码都代表 A,但编码值不同。具体选择哪一个,要视通信双方的"约定"(也可以理解为商量好的协议)。若约定按偶校验传输,则传输 01000001,反之则传输 11000001。此时双方都知道最高位(bit_7)是校验位,不是数值本身。

为了表示更多的欧洲常用字符(如德语中的字母 Ü),对标准 ASCII 码进行了扩展。扩展 ASCII 码由 8 位二进制数码组成,这样就可以表示 256 种不同的符号(ASCII 码值在 128～255 之间的字符常用于画图和画线,以及一些特殊的欧洲字符)。

除 ASCII 码外,较常见的西文字符编码还有 EBCDIC 码,用 8 位二进制码表示,可表示 256 个字符。

2. 中文字符的编码

为了使普通中国人也能使用计算机,需要计算机能够处理汉字。相对于西文字符,中文字符的处理要复杂得多。数值和西文字符可以通过键盘直接输入,每个西文字符都有对应的编码,所以也很容易直接输出。但汉字是象形文字,要让计算机能够处理汉字,首先要解决的就是汉字字符的键盘输入问题,之后才是处理和存储。同样,作为象形文字的汉字,在输出时也与西文不同,需要转换为汉字的字型码。因此,汉字字符的编码包括外码、机内码和用于显示输出的字型码和矢量汉字(为描述上的便利,下文中我们权且将字

型码和矢量汉字统称为输出编码)。

1) 汉字外码

汉字外码也称输入码,主要解决如何将每个汉字变成可以直接从键盘输入的代码。目前的汉字输入码主要可分为字音编码、字形编码、形音编码和数字编码4类。常见的如搜狗、全拼、双拼、智能ABC等属于字音编码;五笔字型等属于字形编码;将字形笔画和拼音结合起来就是音形编码,如世纪形音编码等;区位码则是典型的数字编码。中国国家标准化管理委员会于1981年颁布的《国家标准信息交换用汉字编码基本字符集》(简称国标码,代号为GB 2312—1980)中,规定将所有汉字和字符组成一个96×96的矩阵,每个汉字或字符都属于矩阵中的一个点,有对应的行号(称为“区”)和列号(称为“位”),这种编码就是区位码。例如,汉字“啊”在矩阵的第16行第1列,所以其区位码是1601。

2) 机内码

机内码是汉字在计算机中的编码。主要有国标码、BIG5码(主要在中国台湾和香港地区使用)等。汉字的字符集非常大。国标码GB 2312—1980中共收集了6763个汉字和682个非汉字符号(外文、字母、数字、各种图形等),每个汉字对应一个国标码。

由于汉字数量较大,国标码规定每个汉字机内码都用2字节表示,其编码方法是:高字节和低字节分别在区码和位码的基础上加上二进制数10100000。即

机内码高8位=区码+10100000,机内码低8位=位码+10100000

例如,汉字“啊”的区位码用十进制表示是1601,其对应的二进制数区位码是00010000 00000001。按照上述机内码的编码方法,在高字节和低字节都分别加上10100000,则“啊”的机内码为10110000 10100001。

由上述编码规则可以看出,汉字国标码每个字节的最高位均为1。这样设计的目的是为了避免与ASCII码的冲突。因此,在计算机中,首位是0的为ASCII码字符,首位是1的为汉字。

用2个字节对字符进行编码,可以产生2^{16}个编码,这个数字已可以编码世界上几乎所有的文字。因此,为了形成一种能涵盖世界不同文字字符的编码,国际标准化组织在ASCII码的基础上创建了一种2字节的通用字符编码Unicode(Universal Multiple Octet Coded Character Set)。Unicode编码是针对各国文字和符号进行的、在计算机上使用的统一字符编码,它为每种语言中的每个字符设定了唯一的二进制编码,以满足跨语言、跨平台进行文本转换、处理的要求。

Unicode可支持欧洲、非洲、中东、亚洲(包括统一标准的东亚象形汉字和韩国象形文字)的主要文字。但由于它需要2字节,比ASCII码要多占用1倍的空间,对于可用ASCII编码的字符,效率就显得较低。为了解决这个问题,就出现了一些中间格式的字符集,它们被称为通用转换格式,即UTF(Unicode Transformation Format)。最常见的UTF格式是UTF-8(8bit UTF),它是一种针对Unicode的可变长度字符编码,用1~4B编码Unicode字符,是另一种可以在计算机中表示汉字的编码。

3) 输出编码

要将汉字输出显示,还必须将机内码转换为输出编码。汉字的输出编码用于表示不同字体的汉字字型,分为字型码和矢量汉字两种。字型码是确定一个汉字字型点阵的代码,字

型点阵中的每个点对应一个二进制位。每个汉字对应一个点阵,再编上代号存入存储器中,这就是字模库。汉字在显示时需要在汉字库中查找汉字字模并以字模点阵码形式输出。

点阵字库的汉字因由若干个点组成,故当字体放大时,点会随之放大,使得字看上去比较粗糙。

【例 2-1】 字模的制作过程。

字模的制作过程是将一个汉字"放在"16×16 的方格内,如图 2-3 所示。假设用 1 表示黑点,用 0 表示白点,则 16×16 的点阵汉字可以用 256 位二进制数表示,占用 256b/8=32B 存储空间。按照 1.1.2 节的介绍,即需要占用 32 个存储单元。图 2-3 中给出了汉字对应的点阵数字化编码。

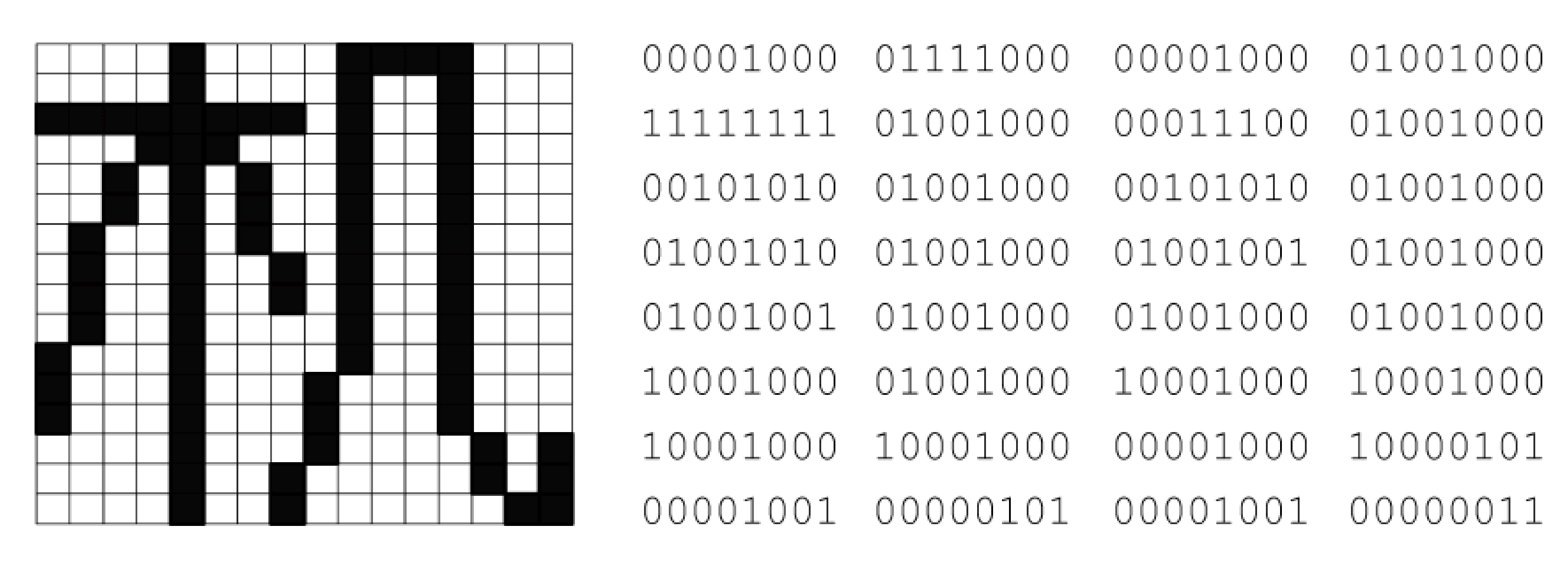


图 2-3 汉字"机"的 16×16 点阵数字化编码

汉字的另一种输出码是矢量汉字。矢量字库保存每一个汉字的描述信息,如一个笔画的起始和终止坐标、半径、弧度等。在显示、打印这一类字库时,需经过一系列的数学运算才能输出结果。矢量字库保存的汉字理论上可以被无限放大,笔画轮廓仍然能保持圆滑清晰。打印时使用的字库均为矢量字库。Windows 使用的字库为以上两类,在操作系统的 WINDOWS\Fonts 目录下,如果字体文件后的扩展名为 FON,表示该文件为点阵字库;若扩展名为 TTF,则表示是矢量字库。

汉字信息从输入到输出的处理过程如图 2-4 所示。

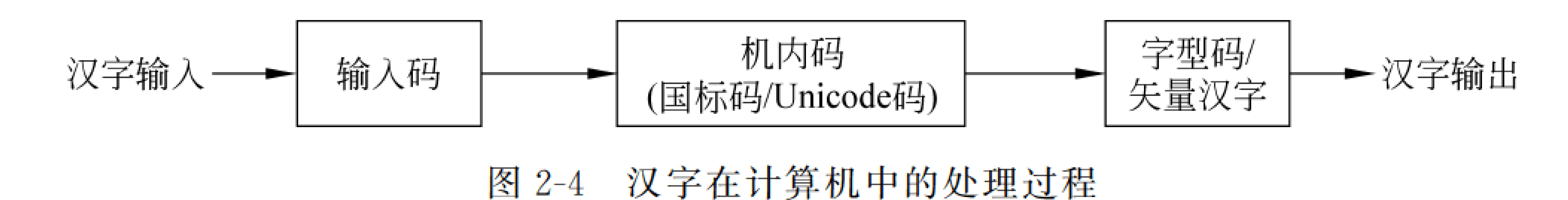


图 2-4 汉字在计算机中的处理过程

2.2.4 声音信息的表示

计算机中存储和处理的信息除数值和文字外,还有各类被称为多媒体的信息,包括声音、图像、视频等。与数值和字符信息不同,这些信息都是连续变化的模拟信号,无法直接用计算机进行存储和处理,必须要首先转换为由 0 和 1 组成的二进制位串,这一过程称为数字化。

1. 声音信息的数字化

声音是通过空气传播的一种连续的波(sound wave,声波),它的连续性体现为:幅值

大小是连续的，可以是实数范围内的任意值；在时间上是连续的，没有间断点（如图 2-5(a)所示），这种在时间和幅值上都连续变化的信号称为模拟信号。相应地，将时间和幅值都不连续的信号称为离散信号（如图 2-5(b)所示）。

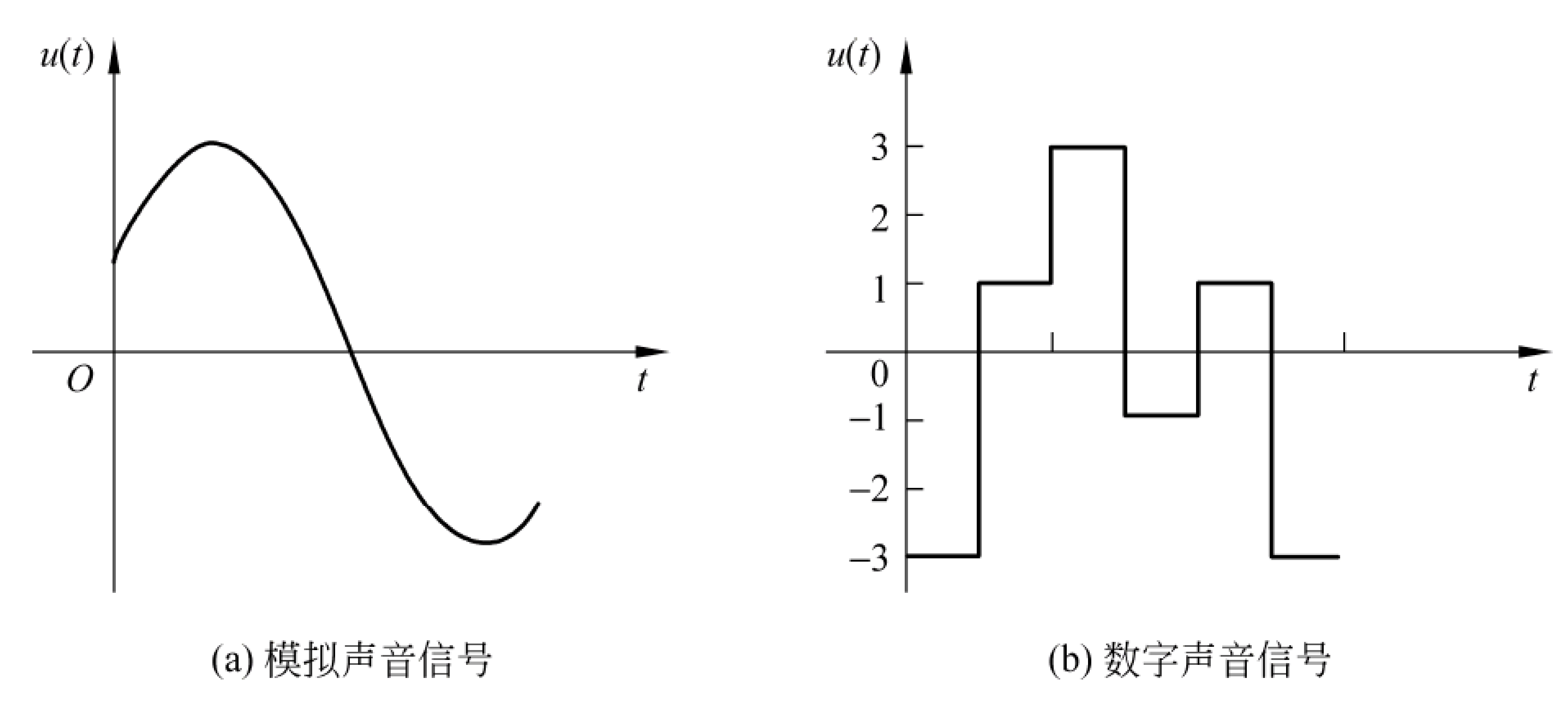


图 2-5　模拟声音信号和数字声音信号

要使连续变化的声音信号能够被计算机处理，首先需要对其进行数字化，变换为数字声音信号（或称数字音频信号）。那么，什么是数字信号呢？数字信号是指在时间和幅值上都离散的信号。所以，声音的数字化过程就是：将时间和幅值均连续变化的模拟声音信号，通过采样（sampling）和量化（measuring），转换为时间和幅值均不连续的离散信号，这种离散的声音信号称为数字音频信号，也就是计算机能够存储和处理的信号。

采样的意思是在某些特定的时刻对模拟信号进行测量，由于测量的时间不连续（如图 2-6 所示的采样时间 $t_1, t_3, \cdots$），所以得到的信号在时间上是离散的，称为**离散时间信号**（discrete time signal）。

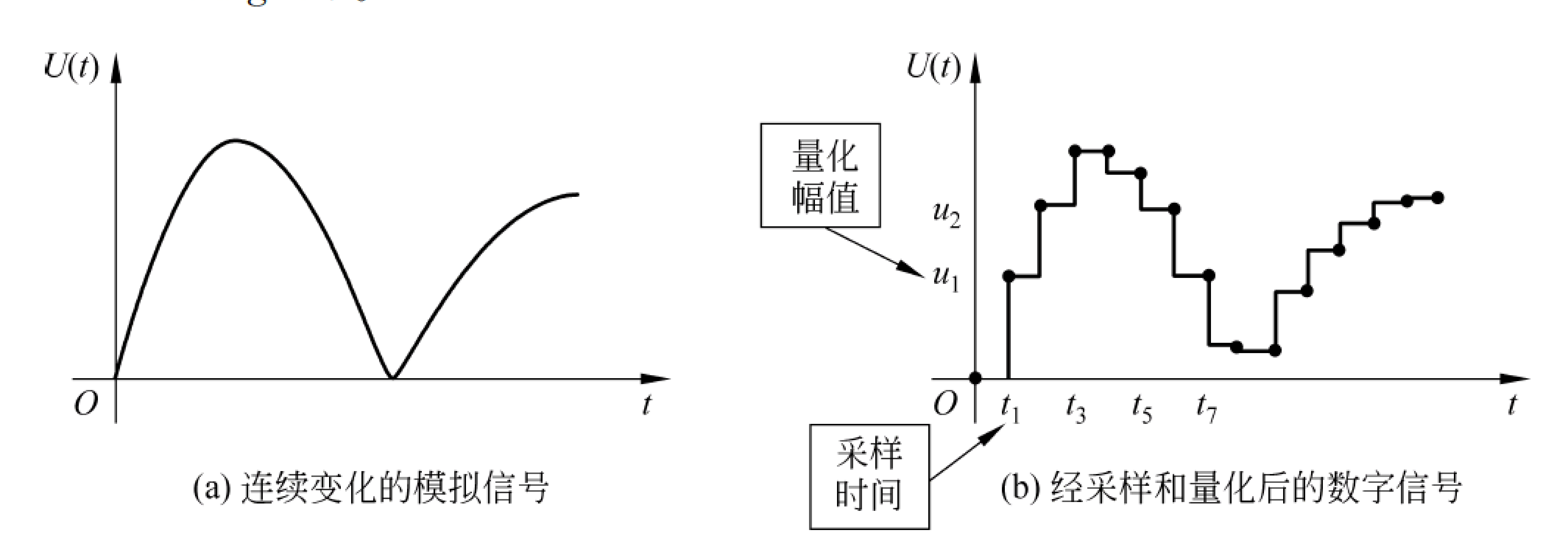


图 2-6　模拟信号的数字化

虽然采样可以实现时间上的离散性，但得到的信号幅值却可以是无穷多个实数值中的一个，即幅值还是连续的（数字信号要求时间和幅值都是非连续的离散值）。因此，还需要对幅值进行离散化，这个过程称为量化。

所谓**量化**，就是把信号幅值的取值数目加以限定，由有限个数值组成，形成**离散幅值信号**（discrete amplitude signal）。例如，假设输入电压的范围是 0～5V，要求它的取值限定为 0V，0.5V，1.0V，…，5.0V 共 11 个值。那么，如果采样得到的幅值是 0.528V，则近

似取值为 0.5V,而如果采样得到的幅值是 0.71V,则取值就近似为 1.0V。如此,就实现了幅值的离散化。经采样和量化后,就得到了时间和幅值都离散的数字信号,如图 2-6 中离散的幅值点。

2. 数字声音信号的编码

由图 2-6 可以看出,若采样的次数越多,表示采样所得幅值的数据位数越长,则数字化后的信号与模拟信号的接近度就越好。例如,上文中的电压量化值不是用 1 位小数位,而是用 2 位小数位表示,则 0.528V 就可以近似为 0.53V,比起用 0.5V 近似,显然其精确度就更高。由于计算机采用二进制,所以数字声音的量化值都是用二进制表示。

单位时间里的采样次数称为**采样频率**(sampling frequency),将量化值的取值个数称为量化级别,而将表示量化级别的二进制数位数称为**采样精度**(sampling precision),也叫**样本位数**或**位深度**,用位(bit)表示。显然,采样频率越高,样本位数越多,声音的质量越高,当然,需要的存储空间也就越多。

与计算机对字符信息的处理类似,数字声音信号在计算机中也需要进行编码。

编码(coding)是将采集到的物理量转换为在计算机中表示的代码的过程。计算机中的数字声音并不是以采集到的声音的真正幅值存储的,而是这些幅值的代码,即编码。数据编码实际上是一种数据的变换过程。

【例 2-2】 数据编码举例。

设量化值为 20、30、40、…、170(间隔 10),共 16 种不同的数据。

16 种量化值相当于 16 种符号,可以用 4 位二进制数表示。即用 4 位二进制数对这 16 个幅值进行编码,如表 2-3 所示。

表 2-3　16 个量化值的二进制编码

量化值	对应编码	量化值	对应编码	量化值	对应编码	量化值	对应编码
20	0000	60	0100	100	1000	140	1100
30	0001	70	0101	110	1001	150	1101
40	0010	80	0110	120	1010	160	1110
50	0011	90	0111	130	1011	170	1111

这种编码方式表示每种符号使用的二进制位数是相等的,称为**自然码编码**。其他常用的编码方法还有如哈夫曼编码(Huffman coding)、算术编码(arithmetic coding)等。

数字声音在计算机存储器中的存放形式称为声音文件格式。相同的数据,可以有不同的存放形式,所以也就有多种文件格式,不同的格式其文件的扩展名不同(如 WAV),每种格式都具有特定的应用场合。计算机中广泛应用的数字化声音文件有两类,一类是采集各种声音的机械振动得到的数字文件(也称波形文件),其中包括音乐、语音及自然界的效果音等,另一类是专门用于记录数字化乐声的 MIDI 格式文件。常见的波形声音文件格式有 WAV、MP3、RealAudio 等。

声音文件的顺利播放,取决于播放器(播放声音的软件)能否正确识别相应的文件格式。一种声音文件可以由一种以上的播放器播放,一种播放器也能播放多种声音文件。

2.2.5 图像信息的表示

俗话说“百闻不如一见”，人类从自然界获取的信息中，视觉信息占了极大的比重。有些花费很多笔墨也很难表达清楚的事物，若用一幅图像描述，可以做到“一目了然”。比如一本好的家电或设备的使用说明书中，总是在文字说明的同时配有详细的操作示意图，阅读这些简图就比较容易理解相应的文字说明，并大致了解设备的基本构造和使用方法。因此，图像也是计算机处理的重要信息类型。

图像(image)是自然界的景物通过人们的视觉器官在大脑中留下的印象。常见的各种照片、图片、海报、广告画等均属于图像。图像可以是简单的黑白图像，也可以是全真色彩的照片。最简单的图像是单色图像(二值图像)，所包含的颜色仅有黑色和白色两种。彩色图像包含了各种色彩(颜色)。

1. 图像信息的数字化

日常生活中看到的图像都是色彩(或灰度)连续变化的模拟图像(比如用胶卷拍出的相片就是模拟图像)，模拟图像的特点是空间上是连续的，可以洗一寸的照片也可以洗二寸的照片，不影响视觉效果。与声音信号一样，要使图像能为计算机所处理和存储，必须将其离散化，即转换为数字图像。

与声音数字化类似，图像的数字化过程也包括采样、量化和编码。不同的是，由于图像是在二维空间坐标上连续变化的函数，对图像来讲：

- **采样**是在空间上将一幅连续图像变换为 $f(x,y)$ 坐标中的一个个点，称为像素点。每个像素点具有颜色空间中的某一种颜色(灰度值)。此时每个像素点的颜色还是连续的。
- **量化(整量)**是用有限位二进制数来表示某个像素点的灰度值(也就是幅值的离散化)。所用的二进制数位越长，可以表示的灰度等级就越多。如果仅用一位二进制码表示像素点的灰度，该像素点就只有“黑”、“白”两种颜色；若用 4 位二进制码来表示，则该像素点就可以有 16 种不同的颜色(或由黑到白 16 种不同的灰度等级)，相应的图像称为 16 色图像。

例如，图 2-7 是一幅只有黑色和白色的二值图像，所以只需要用 1 位二进制数来表示颜色，假设用 0 表示黑色，用 1 表示白色，则离散化后，就转换为图 2-8 所示的编码，这也就是这幅二值图像在计算机中的存储形式。

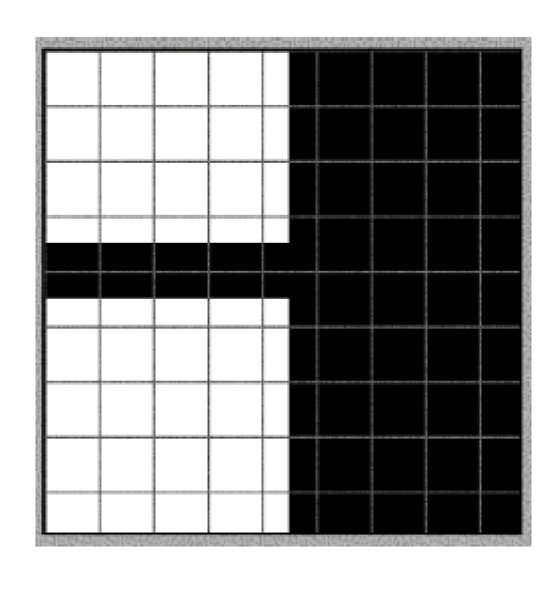


图 2-7　二值图像

1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0

图 2-8　图 2-7 对应的数字图像编码

2. 图像文件格式

图像数据在计算机中的存放形式称为图像文件格式。常用的图像文件格式有位图文件格式(BMP)、索引文件格式(GIF)和JPEG压缩文件格式(JPG)等。

BMP格式是Windows采用的图像文件存储格式,在Windows环境下运行的所有图像处理软件都支持这种格式,能够在任何类型的显示设备上显示。BMP位图文件默认的文件扩展名是BMP或bmp。位图文件是一种不压缩的存储格式,图像质量较高,没有数据损失,但占用的存储空间较大。

GIF(Graphics Interchange Format)格式是CompuServe公司开发的图像文件格式,属于压缩存储方式,因此占用的存储空间很小,在网络中被广泛采用。GIF格式文件还支持透明图像属性和动画图像属性,但表示的颜色数量有限,适合存储颜色较少的卡通图像、徽标等手绘图像。

JPEG(Joint Photographic Experts Group)是由ISO(International Standard Organization,国际标准化组织)和IEC(International Electrotechnics Committee,国际电工委员会)两个组织联合开发的一种算法,称为JPEG算法,又称JPEG标准,相应的文件存储格式为JPG(或jpg)格式。JPEG是一个适用范围很广的静态图像数据压缩标准,既可用于灰度图像,又可用于彩色图像。

JPG文件在压缩时可以调节图像的压缩比和图像保真度,从而根据需要得到不同质量和不同文件大小的图像。JPG格式的文件比较适合存储色彩丰富的照片,虽然数据压缩使图像数据有所损失,但在一定分辨率下,视觉感受并不明显,因此得到了软硬件厂商的普遍支持,几乎所有数字照相机中存放的都是JPG格式的照片文件。

图像严格地讲还可分为静态图像和动态图像。动态图像就是常说的视频,视频文件的存储格式与静态图像文件格式不同,限于篇幅,本书对此不再详述。

除了图像,"图"还包括图形。图形(graphics)也称矢量图,是通过数学公式计算、由程序设计语言实现的图,使用直线和曲线来描述。例如,对于直线,可以通过line,start_point,end_point表示;对于圆,则表示为circle,center_x,center_y,radius;而一幅花的矢量图可以由线段形成外框轮廓,通过设定外框的颜色及外框所封闭的颜色决定花所显示出的颜色(如图2-9所示)。

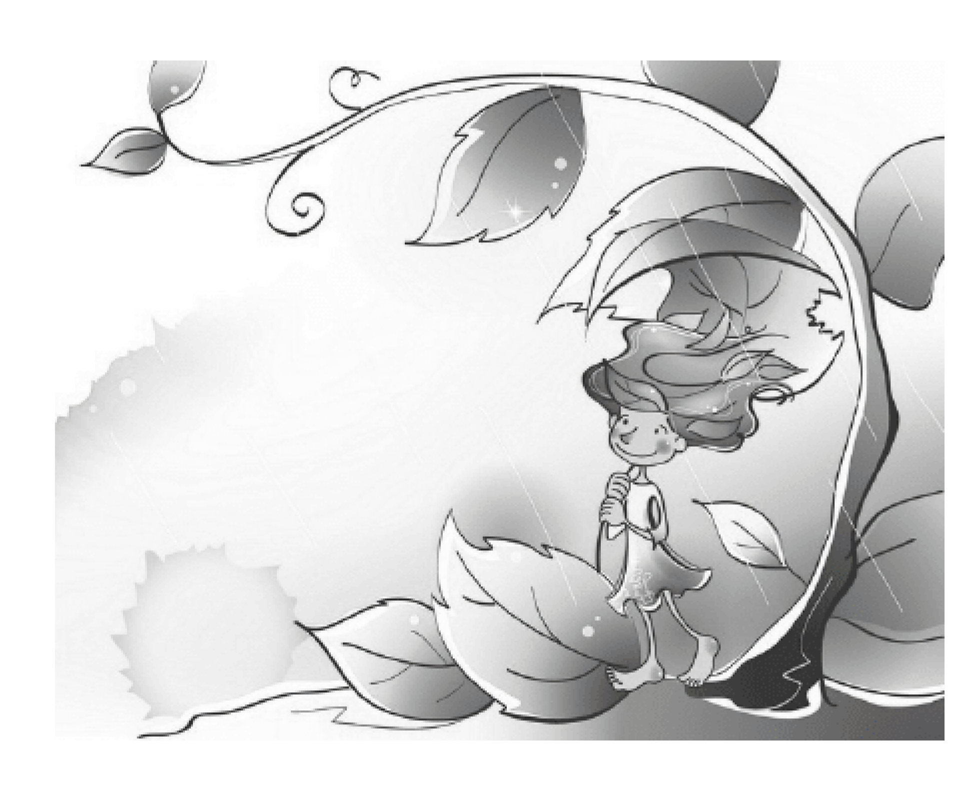


图2-9 矢量图例

有关数字化声音和数字化图像的进一步描述请参阅本书附录C。

为了提高数字化后的声音和图像质量,经采样、量化后的数字音频信号和数字图像通常还需要经过一定的处理,如去噪、锐化等。对音频信号,可以添加各种效果(如淡入淡出、频率均衡和混响等);对图像信号,若需进一步分析,还需要特征提取、图像分割等各种图像处理技术。对此有兴趣的读者可参阅相关书籍。

2.3 计算机中的数制

由 2.1 节的描述已知，**计算机硬件能够直接识别**的只有 0 和 1 构成的二进制码，也就是说，计算机中的数是用二进制表示的。

虽然对计算机来讲，采用二进制有诸多优势。但对人来讲，采用二进制却毋庸置疑存在两个问题：一是在用二进制数表示一个较大的数时，既冗长又难以记忆；二是人类自古就习惯于使用十进制数，对二进制很难接受。

为了解决人与计算机之间存在的这对矛盾，人们编写了自动实现记数制转换的软件。借助软件的辅助，现代计算机也能够**间接理解**十进制以及一些其他进制数了。为了便于后续课程的学习，本节就介绍计算机中常用的几种记数制以及它们相互之间的转换方法。

2.3.1 常用记数制

计算机中的常用记数制，除了计算机唯一能够直接认识的二进制和人类习惯的十进制外，还有一些其他记数制，这些记数制设立的主要目的是为了既在书写上简练，又能与二进制之间有直接的对应关系(即可以直接转换)，如十六进制和八进制。

1. 十进制数

在古人类曾经生活过的岩石洞里发现的刻痕说明人类文明发展的早期就有了计算的需要和能力。考古研究说明，在数的概念出现之后，就出现了数的计算。计算需要借助一定的工具来进行，人类最初的计算工具就是人类的双手，掰着指头数数就是最早的计算方法。一个人天生有 10 个指头，因此十进制就成为人们最熟悉的进制计数法。

十进制(decimal)是人们最习惯、最熟悉的记数制，有 0～9 十个数字符号，用符号 D 标识。一个任意的十进制数可用权展开式表示为

$$\begin{aligned}(D)_{10} &= D_{n-1}\times 10^{n-1}+D_{n-2}\times 10^{n-2}+\cdots+D_1\times 10^1+D_0\times 10^0\\ &\quad +D_{-1}\times 10^{-1}+\cdots+D_{-m}\times 10^{-m}\\ &= \sum_{i=-m}^{n-1} D_i\times 10^i\end{aligned} \tag{2.1}$$

其中，D_i 是 D 的第 i 位的数码，可以是 0～9 十个符号中的任何一个，n 和 m 为正整数，n 表示小数点左边的位数，m 表示小数点右边的位数，10 为基数，10^i 称为十进制的权。

2. 二进制数

二进制(binary)数由 0 和 1 两个符号组成，用符号 B 标识，遵循逢二进位的法则。一个二进制数 B 可用其权展开式表示为

$$\begin{aligned}(B)_2 = B_{n-1}\times 2^{n-1}+B_{n-2}\times 2^{n-2}+\cdots+B_0\times 2^0\\ +B_{-1}\times 2^{-1}+\cdots+B_{-m}\times 2^{-m}\end{aligned}$$

$$= \sum_{i=-m}^{n-1} B_i \times 2^i \qquad (2.2)$$

其中，B_i 为 1 或 0，2 为基数，2^i 为二进制的权，m、n 的含义与十进制表达式相同。为与其他进位记数制相区别，一个二进制数通常用下标 2 或大写字母 B 表示。例如，一个二进制数 1101，可以表示为 1101B，也可以表示为 $(1101)_2$。

【例 2-3】 二进制数 1011.11 可表示为

$$(1011.11)_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-2} = 11.75$$

3. 十六进制数

十六进制(hex)数共有 16 个数字符号：0～9 及 A～F，用符号 H 标识，其运算法则遵循逢十六进位。一个十六进制数 H 的权展开式为

$$\begin{aligned}(H)_{16} &= H_{n-1}\times16^{n-1} + H_{n-2}\times16^{n-2} + \cdots + H_0\times16^0 \\ &\quad + H_{-1}\times16^{-1} + \cdots + H_{-m}\times16^{-m} \\ &= \sum_{i=-m}^{n-1} H_i \times 16^i\end{aligned} \qquad (2.3)$$

式中，H_i 取值范围为 0～F，16 为基数，16^i 为十六进制数的权，m、n 的含义与上面相同。十六进制数也可用下标 16 表示。

【例 2-4】 十六进制数 38EF.A4H 可表示为

$$\begin{aligned}(38EF.A4)_{16} &= 3\times16^3 + 8\times16^2 + 14\times16^1 + 15\times16^0 + 10\times16^{-1} + 4\times16^{-2} \\ &= 14\,575.640\,625\end{aligned}$$

有了二进制和十进制，已经是既满足了计算机的要求，又满足了人的要求，那为什么还要引入十六进制呢？主要的原因就是前面提到的既可以缩短书写长度，又与二进制有直接的对应关系。

在 2.1 节中已经讨论过，在二值符号体系中，一根导线上的电脉冲可以有“出现”和“不出现”两种状态，即一位二进制数码可以有两种状态组合；若是两根导线，则两根导线作为一个整体，其电脉冲的“出现”和“不出现”就有 4 种可能(都不出现，其中 1 根出现，都出现)，即两位二进制数码有 4 种状态组合。依此类推，4 位二进制数码就有 16 种组合(也可以这样想：因为 $2^4=16$)。也就是说，可以用 1 位十六进制数来代表 4 位二进制数，这样，书写的长度就缩短为 1/4；同时，对应关系又非常明确：1 位十六进制数恰好可用 4 位二进制数表示，且它们之间的关系是唯一的。所以，在计算机应用中，虽然机器只能识别二进制数，但在数字的表达上，十六进制的应用也很广泛。

4. 其他进制数

除以上介绍的二进制、十进制和十六进制 3 种常用的进位记数制外，计算机中还可能用到八进制(octal)数。八进制数有 0～7 这 8 个数符，用符号 O 标识，其运算规则为逢八进位。其权展开式可参照式(2.4)。

下面给出任一进位制数的权展开式的一般形式。一般地，对任意一个 K 进制数 S，都可用权展开式表示为

$$
\begin{aligned}
(S)_k = & S_{n-1} \times K^{n-1} + S_{n-2} \times K^{n-2} + \cdots + S_0 \times K^0 \\
& + S_{-1} \times K^{-1} + \cdots + S_{-m} \times K^{-m} \\
= & \sum_{i=-m}^{n-1} S_i \times K^i
\end{aligned}
\tag{2.4}
$$

这里，S_i 是 S 的第 i 位数码，可以是所选定的 K 个符号中的任何一个；n 和 m 的含义同上，K 为基数，K^i 称为 K 进制数的权。

需要注意的一点是：在默认情况下，**十进制标识符 D 可省略，而其他进制数则须标明标识符**。即当数字后无标识符时，计算机将默认其为十进制数。例如：

- 1101B 是二进制数，而 1101 则默认为十进制数（这两个数的大小可是有很大差距的）。
- 2014 是十进制数，2014H 就是十六进制数。两数大小相差近 4 倍。
- ABCDH 是十六进制数，但如果在编写程序时忘记在 ABCD 后写 H，则会报错。

2.3.2 各种数制之间的转换

计算机采用的只有二进制数，编写程序时为方便起见又很少直接使用二进制，因此必然存在不同计数制之间的转换问题。需要说明的一点是：在计算机诞生初期，由于所有的程序都是用二进制码编写的，即使十进制数，也是用二进制的形式（即 0 和 1）来表示的①，所以那个时候只使用二进制。之后，随着编译软件的出现及功能日趋强大，程序编写中逐渐开始更多地采用十六进制，特别是十进制，而将转换为二进制的工作“交给”了编译软件。虽然如此，但由于计算机采用二进制这一暂时无法改变的事实，在计算机应用技术中，依然都是以二进制的概念来描述很多问题的。如 2.2.2 节中已提到的“字长”，就是以二进制位为基础的。另外，在很多具体的软硬件设计中，不同的记数制也都会出现和采用。因此，作为进一步学习的基础，需要非常清楚地了解计算机中的数制以及它们相互之间的转换。

1. 非十进制数到十进制数的转换

非十进制数转换为十进制数的方法比较简单，只要将它们按相应的权表达式展开，再按十进制运算规则求和，即可得到它们对应的十进制数。

【例 2-5】 将二进制数 1101.101 转换为十进制数。

解：根据二进制数的权展开式，有

$$
\begin{aligned}
(1101.101)_2 &= 1\times2^3+1\times2^2+0\times2^1+1\times2^0+1\times2^{-1}+0\times2^{-2}+1\times2^{-3} \\
&= (13.625)_{10}
\end{aligned}
$$

【例 2-6】 将十六进制数 64.CH 转换为十进制数。

解：根据十六进制数的权展开式，有

---

① 用 0 和 1 表示十进制数也称为对十进制的编码。最典型的是 BCD(Binary-Coded Decimal)码，它用 4 位二进制数表示十进制的 0～9，使二进制和十进制之间的转换得以快捷地进行。BCD 码常用于会计系统的设计中。

$$(64.C)_{16}=6\times16^1+4\times16^0+C\times16^{-1}=6\times16^1+4\times16^0+12\times16^{-1}=(100.75)_{10}$$

2. 十进制数转换为非十进制数

十进制转换为非十进制（*K* 进制）数时，整数和小数部分应分别进行转换。整数部分转换为 *K* 进制数时采用“除 *K* 取余”的方法。即连续除 *K* 并取余数作为结果，直至商为0，得到的余数从低位到高位依次排列即得到转换后 *K* 进制数的整数部分；对小数部分，则用“乘 *K* 取整”的方法。即对小数部分连续用 *K* 乘，以最先得到的乘积的整数部分为最高位，**直至达到所要求的精度或小数部分为零为止**。

【例 2-7】 将十进制数 115.25 转换为对应的二进制数。

解：

整数部分		小数部分	
115/2＝57………	余数＝1（最低位）	0.25×2＝0.5………	整数＝0（最高位）
57/2＝28………	余数＝1	0.5×2＝1.0………	整数＝1
28/2＝14………	余数＝0		
14/2＝7 ………	余数＝0		
7/2＝3 ………	余数＝1		
3/2＝1 ………	余数＝1		
1/2＝0 ………	余数＝1		

转换结果为

$$(115.25)_{10}=(1110011.01)_2$$

【例 2-8】 将十进制数 301.6875 转换为对应的十六进制数。

解：

整数部分		小数部分	
301/16＝18 ………	余数＝D	0.6875×16＝11.0000………	整数＝$(11)_{10}=(B)_{16}$
18/16＝1 ………	余数＝2		
1/16＝0 ………	余数＝1		

转换结果为

$$301.6875=12D.BH$$

【例 2-9】 将十进制数 301.6875 转换为对应的八进制数。

解：

整数部分		小数部分	
301/8＝37 ………	余数＝5	0.6875×8＝5.5 ………	整数＝5
37/8＝4 ………	余数＝5	0.5×8＝4.0 ………	整数＝4
4/8＝0 ………	余数＝4		

转换结果为

$$301.6875=(455.54)_8$$

3. 非十进制数之间的转换

由于 $2^4=16$,$2^3=8$,故二进制数与十六进制数、八进制数之间都存在特殊的关系。一位十六进制数可用 4 位二进制数来表示,而一位八进制数可用 3 位二进制数表示,且它们之间的关系是唯一的。这就使得十六进制数与二进制数之间、八进制数与二进制数之间的转换都非常容易。由于二进制在书写上的麻烦和冗长,因此,在计算机应用中,虽然机器只能识别二进制数,但在数字的书写表达上更广泛地采用十六进制数或八进制数。

计算机中常用的二进制数、十六进制数、八进制和十进制数之间的关系如表 2-4 所示。

表 2-4　数制对照表

十进制数	二进制数	十六进制数	八进制数	十进制数	二进制数	十六进制数	八进制数
0	0000	0	0	8	1000	8	10
1	0001	1	1	9	1001	9	11
2	0010	2	2	10	1010	A	12
3	0011	3	3	11	1011	B	13
4	0100	4	4	12	1100	C	14
5	0101	5	5	13	1101	D	15
6	0110	6	6	14	1110	E	16
7	0111	7	7	15	1111	F	17

将二进制数转换为十六进制数的方法是:从小数点开始分别向左和向右把整数和小数部分每 4 位分为一组。若整数最高位的一组不足 4 位,则在其左边补 0;若小数最低位的一组不足 4 位,则在其右边补 0。然后将每组二进制数用对应的十六进制数代替,则得到转换结果。

以同样的方法,可实现二进制数到八进制数的转换。即,从小数点开始分别向左和向右将数每 3 位分为一组。若整数最高位的一组不足 3 位,则在其左边补 0;若小数最低位的一组不足 3 位,则在其右边补 0。

相应地,十六进制数或八进制数转换为二进制数时,可用 4 位或 3 位二进制代码取代对应的一位十六进制或八进制数。

【例 2-10】 将二进制数 110100110.101101B 转换为十六进制数。

解:

二进制数	0001	1010	0110	.	1011	0100
	↓	↓	↓		↓	↓
十六进制数	1	A	6	.	B	4

所以有

$$(110100110.101101)_2=1A6.B4H$$

【例 2-11】 将八进制数$(273.64)_O$转换为二进制数。

解：

八进制数	2	7	3	.	6	4
	↓	↓	↓		↓	↓
二进制数	010	111	011	.	110	100

从而得

$$(273.64)_8 = 010111011.110100\text{B}$$

2.4　二进制数的表示和运算

二进制数只有 0 和 1 两个符号，因此，相比于十进制，二进制数的表示和运算都要更加简单。

十进制数有正数和负数，二进制数也同样有正、负数之分。不同的是，十进制是面向人的记数制，数的符号可以用＋和－表示；而二进制是面向计算机的记数制，数的符号只能用 0 和 1 表示，其中，用 0 表示＋，用 1 表示－。

2.4.1　二进制数的表示

在计算机中，用于表示数值大小的数据称为数值数据。计算机可以处理的数值可以是整数，也可以是小数。对整数的处理比较容易，但对小数，就会有些麻烦。比如小数点处于不同的位置，其数的大小会不同。计算机中，对小数点位置的表示有定点表示法和浮点表示法两种。

1. 数的定点表示

定点表示法就是小数点固定在一个规定的位置上。用这种方法表示的数称为定点数。

1）定点小数的表示

定点小数是指将小数点准确固定在数据某个位置上的小数。为方便起见，通常把小数点固定在最高数据位的左边，称为纯小数。如果考虑数的符号，小数点的前边可以再设符号位（称为数符）。纯小数的定点数可表示为

$$X = X_s.X_{-1}X_{-2}\cdots X_{-(n-1)}X_{-n}$$

这里，X_s 是符号位，用于表示数的正负。小数点是人为规定的，X_{-1} 至 X_{-n} 为数值部分（称为尾数），表示数大小。其中，X_{-1}表示最高有效位，X_{-n}是最低有效位。当数值有 n 位时，定点小数所能表示的数值范围（也称表数范围）为

$$-(1-2^{-n}) \leqslant X \leqslant 1-2^{-n} \tag{2.5}$$

式中的 n 也称为字长。

【例 2-12】 采用二进制定点表示法，将＋0.1001111B 和－0.1001111B 表示为 8 位二进制纯小数。

这里，将数的符号位用 0 和 1 表示，0 表示正，1 表示负，则采用 8 位定点表示法，可将这两个二进制小数表示为

数符	尾数						
0	1	0	0	1	1	1	1

数符	尾数						
1	1	0	0	1	1	1	1

定点小数表示法在运算前，需要对原始数据事先进行处理。即需将用到的数先通过合适的"比例因子"转化为纯小数，计算的结果又要再用比例因子折算成真实值。另外，这种方法能表示的数的范围小，精度也较低。

定点小数表示法比较节省硬件，主要用于早期计算机中。现代通用计算机都已能处理和计算多种类型的数值，定点小数表示方法目前主要用于表示浮点数的尾数部分。

2）整数的表示

纯小数是将小数点固定在最高有效位之前，若将小数点固定在最低有效位之后，则此时的数就变成了下面讨论的纯整数。任意一个整数都可表示为

$$X = X_s X_{n-1} \cdots X_1 X_0 \qquad (2.6)$$

同样，X_s 表示符号，后边的 n 位表示数值部分。

与定点小数表示法一样，定点整数在运算前，也需要通过合适的"比例因子"将原始数据转化为纯整数，计算结束后再将结果用比例因子折算成真实值。

对于用 n 位二进制码表示的带符号的二进制数，纯整数所能表示的数值范围为

$$-(2^{n-1}) \leqslant X \leqslant 2^{n-1} - 1 \qquad (2.7)$$

【例 2-13】 将二进制数＋1010111 和－1010111 用定点表示法表示为 8 位二进制纯整数。

按照计算机中对符号位的表示方法，以上两个二进制数可表示为如下纯整数：

数符	尾数						
0	1	0	1	0	1	1	1

数符	尾数						
1	1	0	1	0	1	1	1

计算机往往使用几种不同的二进制位数来表示一个整数，如 8 位、16 位、32 位、64 位等（用多少位二进制码表示一个整数，在一定程度上与计算机的字长有关）。不同位数（字长）的整数，其表数范围不同，占用的存储空间也不一样。例如：

- 若 $n=8$，则 8 位二进制数的表数范围是－128～＋127。占用 1 字节存储空间（即 1 个内存单元）；
- 若 $n=16$，则 16 位二进制数的表数范围是－32768～＋32767。占用 2B 存储空间。

2. 数的浮点表示

所谓浮点数，是指小数点的位置可以左右移动的数据。在十进制中，一个数可以写成多种表示形式。如 58.123，可以写成 0.58123×10^2，0.058123×10^3，58123×10^{-3}，等等。

同样，一个二进制数也可以写成多种表示形式。如 1011.101 可以写成 0.1011101×2^4，也可以写成 0.01011101×2^5，等等。即：一个二进制数可以用如下形式表示：

$$X=\pm 2^E \times F \tag{2.8}$$

式中，E 称为阶码，即指数值，为带符号整数；F 表示尾数，通常是纯小数。用阶码和尾数表示的数称为浮点数，这种表示数的方法称为浮点表示法。

浮点数的一般格式如图 2-10 所示。

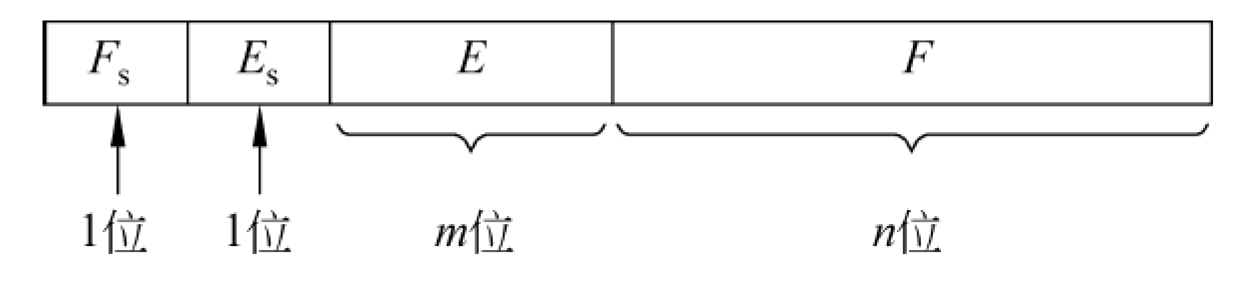


图 2-10　浮点数的一般格式

图中，F_s 为尾符，表示尾数的符号，安排在最高位。它也是整个浮点数的符号位，表示该浮点数的正负；E_s 是阶符，表示阶码的符号，即指数的符号，决定符点数范围的大小；E 是阶码的值，F 是尾数的值。

可以看出，浮点数的表示不是唯一的。当小数点的位置改变时，阶码也会随之改变。这样，同一个数就可以有多种表现形式。为了便于浮点数之间的运算和比较，也为了提高数据的表示精度，规定浮点数的尾数用纯小数表示，即小数点右边第 1 位不为 0，阶码用整数表示。尾数为纯小数、阶码为整数的浮点数称为规格化浮点数。对不满足要求的数，可通过修改阶码并同时左右移动小数点位置的方法使其变为规格化浮点数，这个过程也称为浮点数的规格化。

不论是浮点数还是定点数，在计算机中都要存放在存储器中，而存储器的字长和容量都是有限的。因此，定点数有表数范围，浮点数同样也有。浮点数的表数范围主要由阶码决定，精度则主要由尾数决定。采用图 2-10 格式所表示的规格化二进制浮点数的表数范围为

$$2^{-(1+2^m)} \leqslant |X| \leqslant (1-2^{-n})\times 2^{2^m-1} \tag{2.9}$$

【例 2-14】 将二进制数 1001.101B 表示为浮点数，要求阶码和尾数均为 8 位二进制码。

首先，将 1001.101B 规格化为 $0.1001101\text{B}\times2^4$。这里，阶码 4 的二进制数是 100B。所以，该二进制数的浮点数为

尾符	阶符	阶码								尾数							
0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	0	1

早期的计算机中只有定点数据表示，采用定点数的优点是硬件结构比较简单，缺点是除数据的表示范围比较小之外，主要是必须在运算前将所有参加运算的数据的小数点都对齐到最高位，运算结束后又要再恢复，使运算速度比较慢，也浪费很多存储空间。

随着硬件成本的大幅降低，现代通用计算机中都能够处理包括定点数、浮点数等在内的多种类型的数值。由于浮点数使用阶码和尾数表示数的大小，所以浮点数表示数的范

围比定点数大，相对于定点数来说不易丢失有效数字，提高了运算的精确度。同时，引入浮点数表示法也大幅提高了运算速度[①]。

有关浮点数的进一步描述，请有兴趣的读者参阅其他相关书籍。在从第 5 章开始的 C 语言程序设计的内容中，将会涉及浮点数的应用。

2.4.2 二进制数的算术运算

数字计算机的运算，常常以算术四则运算为基础[②]。和十进制一样，二进制数的算术运算也包含加、减、乘、除四则运算。

1. 加法运算

二进制数的加法运算遵循“逢二进一”法则，具体如下：

0+0=0　　0+1=1　　1+0=1　　1+1=0(有进位 1)

二进制数的加法运算法则也可以这样表述：如果两个相加的数字不同，其和数为 1；如果两个相加的数字相同，其和数为 0。并且进一步有：若两相加数字都为 0，则进位数为 0；若两相加数字都为 1，则进位数为 1。

这样描述二进制数加法运算，就有了“逻辑”的意味了[③]。事实上，计算机的加法运算就是受严格而具有逻辑特性的规则控制的。这一点，随着后续课程的学习，读者将会逐渐理解。

【例 2-15】 求两个二进制数 10110110B 和 01101100B 的和。

解：

```
  1 11111000   进  位
    10110110   被加数
+   01101100   加  数
  ----------
  1 00100010
```

即

$$10110110B+01101100B=100100010B$$

2. 减法运算

二进制数减法法则为

0−0=0　　1−0=1　　1−1=0　　0−1=1(有借位)

该法则同样可以描述为：若两个相减的数字相同，其差值为 0；如果两个相减的数字不同，其差值为 1。并且进一步有：若被减数是 0，则借位数为 1；若被减数是 1，则借位数为 0。

在计算机中，减法运算常常转换为加法运算，这一点将在 2.4.3 节中介绍。

---

① 若采用定点数进行小数运算，一般要借助子程序实现，运算速度要降低两个数量级。

② 引自冯·诺依曼的著作《计算机与人脑》。

③ 有关逻辑的概念将在第 3 章介绍。

【例 2-16】 求两个二进制数 11000100B 和 00100101B 的差。

解：

```
     01111110   借  位
     11000100   被减数
 −   00100101   减  数
 ────────────
     10011111
```

即

$$11000100B-00100101B = 10011111B$$

3. 乘法运算

二进制数乘法与十进制数乘法类似，不同的是二进制数只由 0 和 1 构成，因此其乘法更加简单。法则如下：

$$0\times0=0 \qquad 0\times1=0 \qquad 1\times0=0 \qquad 1\times1=1$$

即，仅当两个 1 相乘时结果为 1，否则结果为 0。运算时若乘数位为 1，就将被乘数照抄加于中间结果，若乘数位为 0，则加 0 于中间结果，只是在相加时要将每次中间结果的最后一位与相应的乘数位对齐。

【例 2-17】 求两个二进制数 1100B 与 1001B 的乘积。

```
          1 1 0 0     被乘数
   ×      1 0 0 1     乘  数
   ──────────────
          1 1 0 0  ┐
        0 0 0 0    │
      0 0 0 0      ├  部分积
    1 1 0 0        ┘
 ────────────────
    1 1 0 1 1 0 0     乘  积
```

运算结果为

$$1100B\times1001B=1101100B$$

从上述运算可以看出，从乘数的最低位算起，凡遇到 1，相当于在最终结果上加上一个被乘数，遇到 0 则不加；若乘数最低位是 1，被乘数直接加在结果的最右边；若次低位是 1，应左移一位后再相加；若再次低位是 1，应左移两位后再相加……依此类推。最后将移位和未移位的被乘数加在一起，就得到两数的乘积。这种将乘法运算转换为加法和移位运算的方法就是计算机中乘法运算的原理。

当然，以上描述都没有考虑数的符号，也就是说都是指正数的乘法。当乘数与被乘数都有正或负时，则相乘的结果就要考虑符号性质了。有关符号数的概念将在 2.4.3 节中介绍。

4. 除法运算

二进制数的除法是乘法的逆运算，其方法和十进制一样。由于除数不能为 0，所以二进制数除法运算的规则是

$$0\div0=0 \qquad 0\div1=0 \qquad 1\div1=1$$

【例 2-18】 求两个二进制数 100111B 与 110B 的商。

```
        110.1
    ┌───────────
110 ╯ 100111
      ──────────
        110
      ──────────
        0111
      ──────────
         110
      ──────────
         00110
      ──────────
            110
      ──────────
              0
```

与十进制数除法运算类似，二进制数的除法也是采用试商的方法求商数。分析上例的过程，可得出二进制数的除法运算可转换为减法和右移运算。

2.4.3 机器数的表示和运算

不论是采用定点表示法还是浮点表示法，计算机中存储和处理的二进制数可通称为机器数。上文已提到，与十进制数一样，二进制数同样有正数和负数。由于二进制是面向计算机的记数制，而计算机只能识别 0 和 1，所以，机器数的正和负分别用 0 和 1 来表示。

机器数总体上可以分为两大类：无符号数和有符号数。

1. 无符号数

所谓无符号数，即没有符号位(也可以简单理解为都是正数)。这种情况下，数中的每一位 0 或 1 都是有效的或有意义的数据。例如，10010110B 是一个二进制数，该数中的每一位都是有意义的，由式(2.2)的权值表达式可以得出其对应的十进制数值为 150。

无符号 n 位二进制纯整数的表数值范围为

$$0 \leqslant X \leqslant 2^n - 1$$

不带符号的二进制码可以被看作一串二进制位的某种组合，如一个字符的编码。

2. 有符号数

有符号数的含义是：该数具有正或负的性质。与十进制数不同的是，计算机中的机器数需要用 0 表示正，用 1 表示负。此时，数据的最高位不是有意义的数据，而是符号位。以 8 位字长为例，D_7 位是符号位，$D_6 \sim D_0$ 为数值位；若字长为 16 位，则 D_{15} 为符号位，$D_{14} \sim D_0$ 为数值位。这样，有符号数中的有效数值就比相同字长的无符号数要小，因为其最高位代表符号，而不再是有效的数值位。符号位数值化的数称为机器数，例如，00010101 是机器数，表示正数；10010101 是机器数，表示负数。

把原来的数值(数据本身)称为机器数的真值(也可以理解为绝对值)，如＋0010101 和－0010101。

符号数的一般格式为

$$\text{符号位} + \text{数值} \tag{2.10}$$

需要注意的是：

- 有符号数最高位的0或1表示的是该数的性质(正数或负数)，最高位不再是数据本身。
- 根据有符号数的不同表示方法，式(2.10)中的“数值”不一定是原始的数据本身，即不一定是真值。

机器数的表示方法有3种，即原码、反码和补码。

1）原码

在原码表示法中，符号位后的就是数值本身。表达形式如下：

符号位＋真值

一个数 X 的原码可记为$[X]_{原}$。**在原码表示法中，不论数的正负，数值部分均为真值。**

【例 2-19】 已知真值 $X=+42$，$Y=-42$，求$[X]_{原}$和$[Y]_{原}$。

解：因为$(+42)_{10}=+0101010B$，$(-42)_{10}=-0101010B$，根据原码表示法，有

$$[X]_{原}=\underline{0}\ \underline{0101010} \qquad [Y]_{原}=\underline{1}\ \underline{0101010}$$

↑ 符号位　↑ 数值部分　　↑ 符号位　↑ 数值部分

注意，根据原码的定义，可以发现一个有趣的现象，即真值0的原码可表示为两种不同的形式：＋0和－0。以8位字长数为例：

$$[+0]_{原}=00000000$$

$$[-0]_{原}=10000000$$

作为数字基准的0在表达形式上不唯一，这是原码表示法的一大缺点。

原码的性质总结如下：

(1) 在原码表示法中，机器数的最高位是符号位，0表示正号，1表示负号，其余部分是数的绝对值，即$[X]_{原}=$符号位$+|X|$。

(2) 原码表示中的0有两种不同的表示形式，即＋0和－0。

(3) 原码表示法的优点是简单、易于理解，与真值间的转换较为方便。它的缺点是除了0的表示不唯一之外，还有在进行加减运算时较麻烦，不仅要考虑是做加法还是做减法，而且要考虑数的符号、绝对值大小及运算结果的符号，这使运算器的设计较为复杂，并降低了运算器的运算速度。

若二进制数 $X=X_{n-1}X_{n-2}\cdots X_1X_0$，则原码表示的严格定义是

$$[X]_{原}=\begin{cases} X & 2^{n-1}>X\geqslant 0 \\ 2^{n-1}-X=2^{n-1}+|X| & 0\geqslant X>-2^{n-1} \end{cases} \tag{2.11}$$

2）反码

反码是原码基础上的变形。对正数来讲，反码的表示方法与原码相同，即最高位为0，其余是数值部分。但**负数的反码表示与原码不同**，其最高位依然是符号位，用1表示，但其余的数值部分不再是原来的真值，而是将真值的各位按位取反。即原先为0的变为1，为1则变为0。真值 X 的反码记为$[X]_{反}$，可用下式表述：

若 $X\geqslant 0$　$[X]_{反}=[X]_{原}$

若 $X<0$　$[X]_{反}=[X]_{原}$的符号位不变，数值部分按位取反

【例 2-20】 已知真值 $X=+42$，$Y=-42$，求 $[X]_{反}$ 和 $[Y]_{反}$。

解：$X=(+42)_{10}=+0101010B$，$Y=(-24)_{10}=-0101010B$，根据反码表示法，有

$[X]_{反}=00101010$ （对正数：$[X]_{反}=[X]_{原}$）

$[Y]_{反}=11010101$ （对负数：$[Y]_{反}=[Y]_{原}$的符号位不变，数值部分按位取反）

由本例可以看出，**对一个用反码表示的负数，其数值部分不再是真值**。

在反码表示法中，同原码一样，数 0 也有两种表示形式(以 8 位字长数为例)：

$$[+0]_{反}=00000000$$

$$[-0]_{反}=11111111$$

反码的性质如下：

(1) 在反码表示法中，机器数的最高位是符号位，0 表示正号，1 表示负号。

(2) 同原码一样，反码中数 0 的表示也不唯一。

若二进制数 $X=X_{n-1}X_{n-2}\cdots X_1X_0$，则反码表示的严格定义是：

$$[X]_{反}=\begin{cases} X & 2^{n-1}>X\geqslant 0 \\ (2^n-1)+X & 0\geqslant X>-2^{n-1} \end{cases} \tag{2.12}$$

在原码表示法和反码表示法中，数值 0 的表示都不唯一，且运算器的设计比较复杂。因此目前在微处理器中已较少使用这两种表示方法(原码表示法主要用于浮点数中的阶码表示)。

目前在计算机中，符号数的表示更多采用的是补码。下面就来详细介绍有关补码的概念和应用。

3. 补码的表示及运算

补码由反码演变而来，其定义为：对正数，补码与反码和原码的表示方法相同，即最高位为 0，其余是数值部分。但**负数的补码表示与原码和反码不同**，其最高位的符号位不变，但其余的数值部分是反码的数值部分加 1，即将原码的真值按位取反再加 1。

真值 X 的补码记为$[X]_{补}$。可用下式表述：

$$若\ X\geqslant 0 \quad [X]_{补}=[X]_{反}=[X]_{原}$$

$$若\ X<0 \quad [X]_{补}=[X]_{反}+1$$

【例 2-21】 已知真值 $X=+42$，$Y=-42$，求 $[X]_{补}$ 和 $[Y]_{补}$。

解：因为 $X>0$，所以

$$[X]_{补}=[X]_{反}=[X]_{原}=00101010$$

因为 $Y<0$，所以

$$[Y]_{补}=[Y]_{反}+1=11010101+1=11010110$$

不同于原码和反码，数 0 的补码表示是唯一的。仍以 8 位字长数为例，由补码的定义知：

$$[+0]_{补}=[+0]_{反}=[+0]_{原}=00000000$$

$$[-0]_{补}=[-0]_{反}+1=11111111+1=\boxed{1}\ 00000000$$

↓

自然丢失

即对 8 位字长来讲，最高位的进位因超出字长范围，会自然丢失，所以

$$[+0]_{补}=[-0]_{补}=00000000$$

事实上，补码的概念在日常生活中也常见到。如钟表，若要从 9 点拨到 4 点，可以有两种拨法：

(1) 逆时针拨到 4 点：9－5＝4。

(2) 顺时针拨到 4 点：9＋7＝4。

两个方向都能拨到 4 点，是因为在时钟系统中有 12 这个最大数，它称为该系统的模，它是自然丢失的。对时钟系统的模 12 而言，9－5＝9＋7，7 称为－5 的补数。所以，－5 的补数可用下式得到：

$$(-5)_{补}=12-5=7$$

即

$$9-5=9+(-5)=9+(12-5)=9+7=12+4=4$$

↓

模，自然丢失

由此可见，引入补码可以将减法运算转换为加法运算。可以将此概念推广到整个二进制系统。二进制记数系统的模为 2^n，这里的 n 表示字长。

【例 2-22】 设字长 $n=8$，用补码的概念计算 96－20。

解：因为 $n=8$，故模为 $2^8=256$。则有

$$96-20=96+(-20)=96+(256-20)=96+236=256+76=76$$

↓

模

即，在模为 2^n 的情况下，96－20＝96＋236。

－20 的二进制表示为 11101100，该数正好是十进制的 236。这样，就利用了负数的补码概念，将减法运算转换成了加法运算。

利用补码实现加减运算的规则如下。

补码的加法规则：$[X+Y]_{补}=[X]_{补}+[Y]_{补}$。

补码的减法规则：$[X-Y]_{补}=[X]_{补}-[Y]_{补}=[X]_{补}+[-Y]_{补}$。

【例 2-23】 已知真值 $X=+0110100B$，$Y=-1110100B$，求 $[X]_{补}+[Y]_{补}$。

解：这里 $X>0$，所以

$$[X]_{补}=00110100B$$

$Y<0$，所以

$$[Y]_{补}=[Y]_{反}+1=10001011B+1=10001100B$$

由补码的加法运算规则知：

$$[X+Y]_{补}=[X]_{补}+[Y]_{补}=00110100B+10001100B=11000000B$$

【例 2-24】 设 $X=+51$，$Y=+66$，求 $[X-Y]_{补}$。

解：由补码的减法运算规则 $[X-Y]_{补}=[X]_{补}+[-Y]_{补}$，先求出 $[X]_{补}$ 和 $[-Y]_{补}$：

$$X=(+51)_{10}=(+0110011)_2,\quad [X]_{补}=00110011B$$

$$-Y=(-66)_{10}=(-1000010)_2,\quad [-Y]_{补}=10111110B$$

然后求$[X]_{补}+[-Y]_{补}$：

$$\begin{array}{r} 00110011 \\ +\ 10111110 \\ \hline 11110001 \end{array}$$

所以

$$[X-Y]_{补}=11110001B$$

由补码运算规则知，两补码相加的结果为和的补码。以上两例的运算结果的符号位为1，表示结果为负数。按照补码的定义，有“负数的补码＝其原码按位取反＋1”，而原码的定义是“符号位＋数值”。所以，当补码数的最高位为1时，表示该数是负数，即此时符号位后的数值“不是真的数值”，需要将其“还原”，即将数值部分按位取反加1，得出真值。

所以，对例2-24的运算结果$[X-Y]_{补}=11110001$，符号位用－表示，数值部分按位取反加1，就得到：$X-Y=-0001111B=-15$。

所以，在计算机中引入补码的主要目的就是将减法运算转换为加法运算。另外，由上述分析已知，在补码表示法中，数0的表示是唯一的。因此，在微机中，凡涉及符号数都是用补码表示的。

*2.5 计算机中信息处理的一般过程

在信息时代，要使各种信息能够最大限度地发挥作用、为人类所用，必须利用计算机这个信息时代最为重要的工具。只有通过计算机高效的处理，信息才能真正在广泛的领域中实现“可用”。例如，只有通过计算机对大量气象数据信息进行的快速分析和计算，才能有准确的天气预报；各种商务网站因为有计算机的管理，才有可能为用户提供网上购物的服务，等等。

利用计算机实现对信息的处理和利用，需要经过以下过程，即信息的采集、信息的表示和压缩、信息的存储和组织，信息的传输、信息的发布和检索。

2.5.1 信息采集

要利用计算机对信息进行处理，须将现实生活中的各类信息转换成智能机器能识别的符号，然后才能再加工处理成新的信息。将信息转换成具体的符号就是数据，可以说数据是信息的符号化，是信息的具体表示形式。数据可以是文字、数值、声音、图像和视频等。

对文字和数值信息的采集方法很多，传统的是键盘输入，随着技术的发展，现在语音输入、手写输入、扫描加模式识别等输入方法也日渐普及。

声音、图像和视频信息也常简称为多媒体信息。对这类信息的采集方法也很多，常用的如录音笔、数码照相机、数码摄像机等。

需要明确的一点是，不论是哪种设备和方法，它们所采集的各类信息最终在计算机中

都必须转换为计算机能够识别和处理的、由 0 和 1 组成的二进制码，而这些二进制码可以通称为数据。

我们已经知道，二进制的 0 和 1，在计算机中实际上是“低电平”和“高电平”。所以，一串二进制码数据事实上是一串电脉冲信号，或者说信号是数据的电磁或光脉冲编码，是各种实际通信系统中适合信道传输的物理量。

“信息”、“数据”和“信号”这 3 个名词，严格地讲，是 3 个不同的概念。但在基于计算机的信息处理中，它们相互间是有关系的。信息的采集需要通过信道、以信号的形式输入到系统，再转换为具体的符号，也就是数据，进行处理。

2.5.2 信息表示和压缩

计算机可处理的信息包括数值、文字、声音、图形图像和视频。1.3 节中对除视频之外的各类信息的表示都已做了简要的描述。

信息采集到计算机中后需要存储。虽然现代计算机中存储设备的容量在不断增大，但受信道传输效率、硬盘容量、处理器速度等各种因素的限制，在很多情况下，仍然希望能够在尽可能小的数据量中包含尽可能多的信息。

通常，多媒体信息的数据量都较大。例如，一张写满文字的 A4 纸的数据量约为 50KB，一幅 3264×2448 的未经压缩的照片的数据量约为 4MB，1 分钟的 MPEG1 压缩视频大约需要 10MB 的存储空间。庞大的数据量造成传输和存储的不便，因此，需要对采集的信息进行压缩。

压缩的任务就是在保持信源信号在一个可以接受质量的前提下，把需要的数据量（比特数）减到最低程度，以减少存储和传输的成本。

有关数据压缩的详细介绍请参阅相关专业书籍。

2.5.3 信息存储和组织

今天的时代被称为信息爆炸时代，通过网络等各种媒介人可以获得大量的各类信息。那么，我们是否考虑过这些信息、特别是电子信息是如何存放的？我们又为什么可以那样快速地就找到它们？这就是信息的组织。

计算机通过文件和数据库技术来对信息进行组织和管理。文件是指存放于计算机中、具有唯一文件名的一组相关信息的集合。计算机中所有的信息，包括各种不同类型的程序都是以文件的形式存放的。文件包括有结构文件和无结构的流式文件两种类型。流式文件指由字符序列集合组成的文件，例如一个源程序文件。

有结构文件（如图 2-11 所示）由一条条记录组成，比如一件商品的名称、规格、生产厂家、价格等，就可以形成一条记录。

一组同类的记录可以形成一个文件，一组相关的文件可以形成数据库。

文件系统使用方便，但存在很多缺陷，如数据冗余（同样的数据出现在多个文件中）、数据不一致性（当一个文件中数据被修改时，另一个文件中的相同信息没有同步修改）、安

商品编号	商品名	规格	定价	生产厂
100001	圆头螺钉	M4	¥1.20	西安金属材料厂
100002	方头螺钉	M6	¥2.50	西安金属材料厂
100003	六角螺钉	M6	¥3.00	西安金属材料厂
200001	螺母	M4	¥0.60	西安金属材料厂
200002	螺母	M6	¥0.80	西安金属材料厂
200003	六角螺母	M6	¥1.80	西安金属材料厂

图 2-11　有结构文件示例

全性差等。

为了弥补传统文件系统的不足，诞生了数据库(Database，DB)技术。就像仓库是用于存放产品一样，数据库中存放的是就是大量的数据。仓库中的存放的产品会按类型、用途、生产厂等分门别类排放在不同的地方，并且由统一的仓库管理员进行相应的入库和出库登记。数据库中的数据也按一定的规则存放和管理，并为不同的用户提供需要的数据服务和信息共享。对应于仓库管理员，数据库的管理由数据库管理系统(Database Management System，DBMS)完成。它是一种软件产品，对数据库中的数据进行集中有效的管理，并为应用程序提供数据资源访问。事实上，我们之所以能够通过网络查询到各种需要的信息，除搜索引擎(如百度)外，还要依赖于各个 DBMS 从数据库中去找到相应的数据。

数据库与文件系统最主要的不同是，数据库中的数据是相关的。例如，一位在校学生可能有个人学籍信息、健康信息、选课信息等，这些信息会出现于不同的表格，存放在不同的部门，当某学生中途离校，该生的学籍信息会取消，同时其他所有表格中该生的信息也都会被修改。这一点是文件系统所无法实现的。当然，除此之外数据库技术还克服了上述文件系统所固有的其他缺点。图 2-12 是一个“工厂产品管理数据库”的结构示意图。

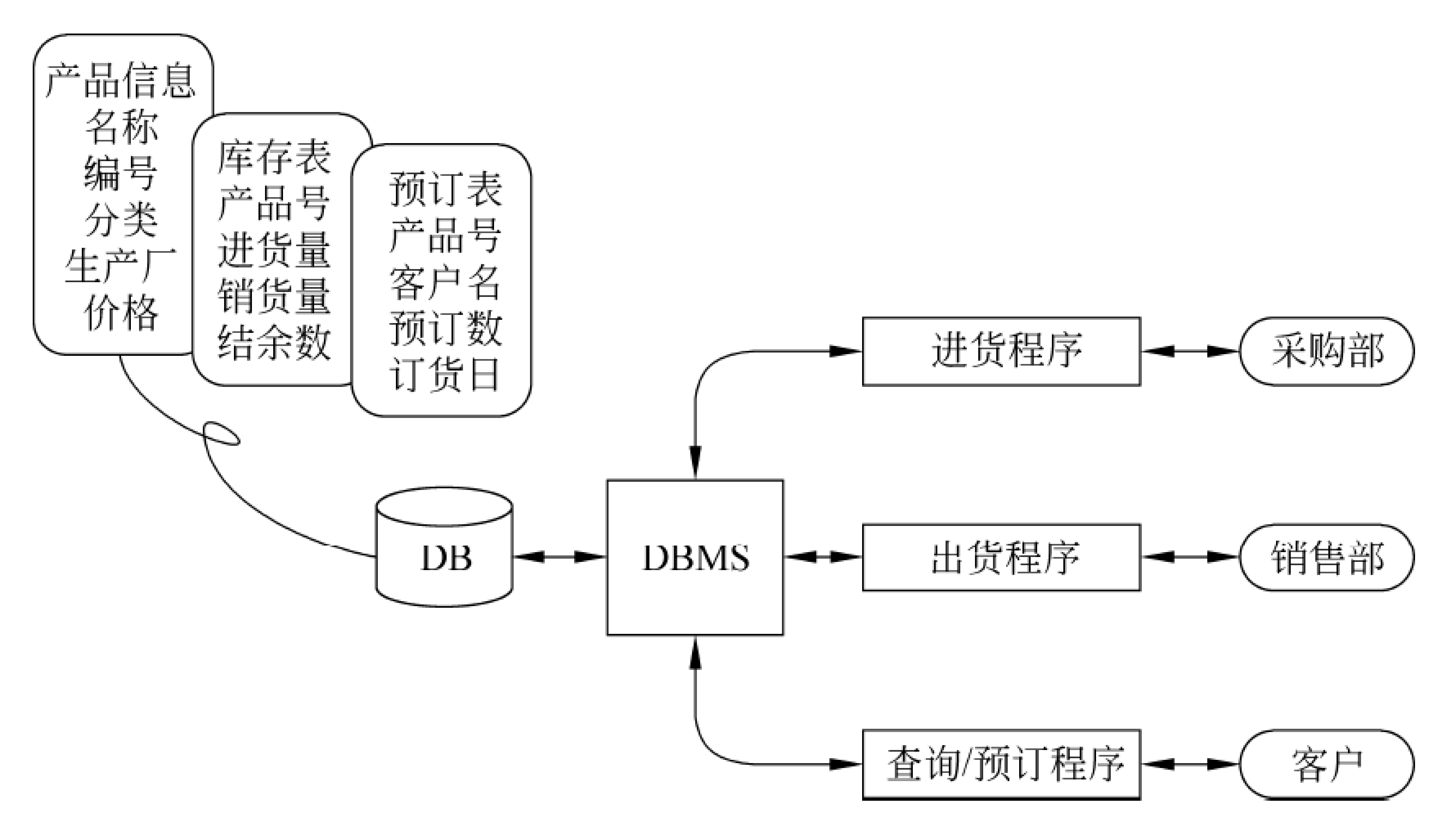


图 2-12　产品管理数据库系统示意图

2.5.4　信息的传输

信息的传输需要通过互联网。今天，我们可以通过电子邮件、即时通信等方法与世界各地的朋友取得联系；为了获取知识，可以利用搜索引擎将存放在世界各地不同数据库中

的信息反馈到查询者的计算机屏幕上。这些都需要基于网络技术。

计算机网络源于计算机与通信技术的结合,它利用各种通信手段,例如电话线、同轴电缆、无线电线路、卫星线路、微波中继线路、光纤等,将地理上分散的计算机有机地连在一起,相互通信而且共享软件、硬件和数据等资源。网络技术始于20世纪50年代,自诞生至今发展迅猛,由主机与终端之间的远程通信,发展到今天世界上成千上万台计算机的互联,形成了遍布全球的互联网络(Internet),从而使网络成为生活中不可或缺的一部分。

有关计算机网络的基础知识将在第4章中介绍。

2.5.5 信息检索

在科学研究中需要查阅相关文献时,当希望到网上商城去"淘"一点价廉物美的商品时,搜索引擎提供了强大的信息检索功能,实现从海量数据中查找到所需要的信息。这就涉及信息检索技术。

信息检索技术是在计算机技术和网络技术的基础上发展和完善起来的,在经历了手工检索、计算机脱机检索、联机检索阶段之后,如今已实现了网络化,并向更高级的智能化发展。

信息检索是指将杂乱无序的信息有序化后形成信息集合,并根据需要从信息集合中查找出特定信息的过程。实现检索的前提(或基础)是要将一定范围内的信息进行筛选、特征描述、有序化处理,形成信息集合,即建立数据库。而检索是采用一定的方法与策略从数据库中查找出所需信息。所以,信息检索也可以简单地理解为信息查找(information search)。

信息检索的实质是将用户的检索标识与信息集合中存储的信息标识进行比较与选择(或称为匹配(matching)),当用户的检索标识与信息存储标识匹配时,信息就会被查找出来,否则就查不出来。匹配有多种形式,既可以是完全匹配,也可以是部分匹配,这主要取决于用户的需要。

在以上讨论的基于计算机的信息处理中,计算机的硬件和操作系统是平台,网络是信息传输和检索的通道,信息的组织、管理及信息的处理等都需要利用计算机程序设计语言去实现。

习　　题

1. 计算机硬件能够直接识别的数制是(　　)。
2. 目前国际上广泛采用的西文字符编码是标准(　　),它是用(　　)位二进制码表示一个字符。
3. 要将模拟声音信号让计算机存储和处理,必须转换为(　　)。
4. "采样"的工作是实现了(　　)的离散化。
5. 采用16位编码的一个汉字存储时要占用的字节数为(　　)。

6. 位图文件的存储格式为(　　),用数码相机拍摄的照片的文件格式一般为(　　)。
7. 若处理的信息包括文字、图片、声音和电影,则其信息量相对最小的是(　　)。
8. 模拟信号是指(　　)都连续变化的信号。
9. 在微机中,信息的最小单位是(　　)。
10. 在计算机中,1B=(　　)b,1KB 表示的二进制位数是(　　)位。
11. 完成下列数制的转换:
(1) 10100110B=(　　)D=(　　)H
(2) 0.11B =(　　)D
(3) 253.25 =(　　)B =(　　)H
(4) 1011011.101B=(　　)O =(　　)H=(　　)D
12. 完成下列二进制数的算术运算:
(1) 10011010+01101110=(　　)
(2) 11001100－100=(　　)
(3) 11001100×100=(　　)
(4) 11001100÷1000=(　　)
13. 写出下列真值对应的原码、反码和补码:
(1) $X=-1110011B$
(2) $X=-71D$
(3) $X=+1001001B$
14. 已知 X 和 Y 的真值,求$[X+Y]_{补}$和 $X+Y$。
(1) $X=-1110111B$　　$Y=+1011010B$
(2) $X=56$　　$Y=-21$
15. 已知 $X=-1101001B$,$Y=-1010110B$,求$[X-Y]_{补}$和 $X-Y$。
16. 请简述计算机采用二进制的理由。
17. 计算机中对信息的组织和管理方式有两种,即(　　)和(　　)。
18. 简述基于计算机的信息处理的一般过程。

第3章 系统软硬件构造

引言

艾伦·图灵奠定了计算机的理论基础，冯·诺依曼则创建了现代计算机的体系结构和基本原理。历经半个多世纪的发展，计算机的功能虽早已今非昔比，但其工作原理和体系结构在总体上依然是冯·诺依曼计算机结构。

也许很多读者没有考虑过今天已认为是理所当然的问题：为什么可以在屏幕上同时打开多个"窗口"？为什么总希望内存越大越好？文件存到了硬盘的什么地方？是怎么放进去的？等等。这一系列问题的答案就隐藏在计算机硬件系统和操作系统工作原理中。

本章从基本逻辑运算及其门电路入手，试图借助推理和"搭积木"的思维模式，去解析系统的"构造"过程，帮助读者理解什么是"抽象"和"封装"，并揭晓上述问题的答案。

教学目的

- 理解逻辑运算及基本逻辑门。
- 理解常用逻辑电路及其构造过程。
- 深入理解冯·诺依曼计算机结构和工作原理。
- 了解图灵机和计算机的关系。
- 了解冯·诺依曼结构的不足。
- 了解操作系统的基本功能。
- 深入理解进程的基本状态及进程的生命周期。
- 理解操作系统中存储器管理的功能。
- 了解文件、文件的组织结构及文件管理的功能。

3.1 逻辑代数基础

在2.1节中，详细讨论了计算机采用二进制的理由。其实，简单的根源就是二进制能够与开关元件的"开"和"关"两种状态一一对应起来。像开关元件这样只有两种可能状态

的器件称为逻辑器件。现代计算机正是由各种各样的逻辑器件组成的,而它的数学基础就是逻辑代数。

逻辑代数的发明人是英国数学家乔治·布尔(George Boole,1815—1864),所以也称为布尔代数,主要研究和判断相关的逻辑运算,被广泛应用于开关电路和数字逻辑电路的分析中。

逻辑代数用字母表示变量(称逻辑变量),只有 0 和 1 两个取值。逻辑代数表示的是逻辑关系,不是数学关系。所以,逻辑运算与算术运算不同,算术运算是将一个二进制数的所有位视为一个数值整体来考虑,低位的运算结果会影响到高位(如进位等);而**逻辑运算是按位进行的运算**,其低位运算结果不会对高位产生影响。即逻辑运算没有进位或借位。

3.1.1 关于逻辑

从哲学的角度讲,逻辑(logic)是反映思维的规律。或者说,是推论和证明的思想过程。逻辑的基本表现形式是命题和推理。

推理是从前提推出结论的思维过程,前提是已知的命题,结论是通过推理规则得出的命题。所以,归根结底,要说清楚逻辑运算,需要先说清楚什么是命题。

1. 命题

所谓命题,简单地讲,就是能判断真假的陈述句。这种陈述句的判断只有两种可能,即正确和错误。命题的判断(可以理解为就是这句话表示的意思)叫作**真值**。真值为正确的命题称为“真”,为错误的命题称为“假”。因此,又可以说命题是具有确定的“真”或“假”的陈述句。比如,“水是无色的”,这是真命题;“冰是有色的”,就是假命题。

【例 3-1】 判断下列语句是否为命题。

(1) 雪是白色的。

(2) 2 是素数。

(3) 5 能被 3 整除。

(4) 明天上午有课吗?

(5) $x+y>8$。

(6) $2+5=7$。

在这 6 条语句中,(4)是问句,不是陈述句,所以不是命题;(1)~(3)和(6)是陈述句,并且具有明确的判断(即正确或错误),所以它们是命题。其中,(1)、(2)和(6)所表达的含义是正确的(我们也说它们的“真值”是“真”的),所以它们是真命题,(3)所表达的含义是错误的(即它们的“真值”是“假”的),所以(3)是假命题;(5)虽然是陈述句,但它不是命题,因为它没有确定的真值,只有当 x 和 y 为给定值后,该语句才能成为命题。

从以上分析可以得出,判断一个句子是否为命题,首先看它是否为陈述句,其次看它的真值是否是唯一确定的。

语句(1)~(3)和(6)都是简单陈述句,不可再分解,因此也称它们为简单命题。

【例 3-2】 观察以下语句，判断是否为命题。

（1）每位德国学生既要学习英语，也要学习法语。

（2）从西安到北京可以乘火车去，或者乘飞机去。

（3）3 不是偶数。

例 3-2 中第（1）句话可以说成"每位德国学生既要学习英语，并且也要学习法语"。这里，"每位德国学生要学习英语"是一个简单命题，"每位德国学生要学习法语"也是一个简单命题，且它们的真值都为真（亦即都是正确的）。这两个简单命题用"并且"联结在一起，表示了一种"同时存在"或"同时为真"的意思。

第（2）句话，用"或者"这个词将"从西安到北京可以乘火车去"和"从西安到北京可以乘飞机去"这两个简单命题联结在一起，表示两者任意一种都可以，或说只要有一个存在就可以（事实上，也不可能一个人同时又坐火车又乘飞机）。

第（3）句话的"不是偶数"也可以说成是"并非偶数"，表示了"是"的反意（将"是"的意思翻转）。

这里的"并且""或者""并非"在命题逻辑中称为联结词。这种用联结词联结起来的命题称为复合命题（例 3-2 中的 3 条语句都是复合命题）。而这些联结词则表示着一种关系或者运算。

由此可以得出，**任何复合命题都可以由简单命题通过联结词所表示的某种运算得到**。

2. 命题的符号化

将一个简单命题用英文字母来表示，称为命题的符号化。例如：

A：雪是黑色的。

B：2 是素数。

P：每位德国学生要学习英语。

Q：每位德国学生要学习法语。

这里，A 是假命题，其余是真命题。

简单命题的真值是确定的，所以真值也可以符号化。通常用 1 表示真，用 0 表示假。另外，联结词也可以符号化。如果将上文提到的联结词"并且""或者""并非"，分别用 and、or、not 来表示，则例 3-2 中的 3 条复合命题就可以符号化为如下形式：

（1）P and Q（P：每位德国学生要学习英语，Q：每位德国学生要学习法语）

（2）R or S（R：从西安到北京可以乘火车去，S：从西安到北京可以乘飞机去）

（3）not T（T：3 是偶数）

可以看出，当命题符号化之后，一个复杂的命题是可以由简单命题通过逻辑运算得到的。在以上符号化后的 3 条语句中：

- and 所表示的"并且"的含义是：同时存在，或同时为"真"。这种关系称为逻辑"与"。
- or 表示的"或者"，是意味着只要有一个存在或者说一个为"真"就可以，这种关系称为逻辑"或"。
- not 表示了"并非"，意为取反（将"3 是偶数"取反为"3 不是偶数"），称为逻辑"非"。

命题是逻辑的基本表现形式，所以，联结词所表示的运算就是逻辑运算。由于逻辑的“真”和“假”，可以符号化为二进制的 1 和 0。因此，逻辑运算也就是 1 和 0 之间的运算。既然如此，逻辑运算和逻辑推理也就可以被计算机处理了。

3.1.2 基本逻辑运算

基本逻辑运算包括“与”、“或”、“非”3 种，在此基础上还可以演变出其他各种复杂的逻辑关系。

1. 逻辑“与”

逻辑“与”的含义是：当全部条件都满足时，结果才会发生。或者可以描述为：当输入条件全部为“真”时，输出的结果为“真”；若输入有一个条件不满足（为“假”），则结果就为“假”。这一点可以用从图 2-2 所示的二极管电路得出。

“与”运算用符号 ∧ 或者 · 表示，遵循如下运算规则：

$$1 \wedge 1 = 1 \quad 1 \wedge 0 = 0 \quad 0 \wedge 1 = 0 \quad 0 \wedge 0 = 0 \tag{3.1}$$

式(3.1)所示规则的含义是：参加“与”操作的两位中只要有一位为 0，则相“与”的结果就为 0；仅当两位均为 1 时，其结果才为 1。

“与”运算相当于按位相乘（但不进位），所以又叫作“逻辑乘”。可以表示为

$$Y = A \wedge B \quad \text{或者} \quad Y = A \cdot B \tag{3.2}$$

式(3.2)的含义是：仅当 A 和 B 全部为“真”时，结果 Y 才为“真”。

对两个多位二进制数来讲，逻辑运算执行按位运算（算术运算是按数据运算），对“与”运算来讲，就是两个数按位相“与”。

【例 3-3】 计算 11011010B∧10010110B。

解：

$$\begin{array}{r} 11011010 \\ \wedge\, 10010110 \\ \hline 10010010 \end{array}$$

即 11011010B∧10010110B=10010010B。

“与”运算通过称为“与门”的逻辑器件实现（详见 3.1.3 节），其逻辑关系反映在电路中，相当于开关的串联（如表 3-2 所示）。

2. 逻辑“或”

逻辑“或”的含义是：在全部输入中，只要有一项条件满足，结果就会发生。“或”运算用符号 ∨ 表示，遵循如下运算规则：

$$0 \vee 0 = 0 \quad 0 \vee 1 = 1 \quad 1 \vee 0 = 1 \quad 1 \vee 1 = 1 \tag{3.3}$$

运算规则表示：参加“或”运算的两位二进制数中，仅当两位均为 0 时，其结果才为 0；只要有一位为 1，则“或”的结果就为 1。该规则还可以表述为：当且仅当输入全部为假时，输出结果才为假。

“或”运算执行两个数按位相“或”的运算，又叫作“逻辑加”，可以表示为

$$Y = A \vee B \quad 或者 \quad Y = A + B \tag{3.4}$$

式(3.4)的含义是：当且仅当逻辑变量 A 和 B 均为 0(假)时，Y 为假；A 和 B 任意一个为 1(真)，则 Y 为真。其逻辑关系可以用表 3-1 的形式表示。由于表中反映了输出 Y(复合命题的真值)和输入 A、B(简单命题的真值)的逻辑关系，所以，该表也称为**真值表**。

表 3-1 “或”逻辑关系真值表

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

【例 3-4】 计算 11011001B∨11111111B。

解：

$$\begin{array}{r} 11011001 \\ \vee\, 11111111 \\ \hline 11111111 \end{array}$$

即 11011001B∨11111111B = 11111111B。

“或”逻辑关系反映在电路中，相当于开关的并联(参见表 3-2)。

同“与”运算类似，“或”运算通过称为“或门”的逻辑器件实现。

思考 试比较二进制数的“或”运算和加法运算的异同。

3. 逻辑“非”

逻辑“非”的含义是：当决定事件结果的条件满足时，事件不发生。“非”运算是按位取反的运算，即，1 的“非”为 0，而 0 的“非”为 1。或者表述为：“真”的“非”为假，反之亦然。

“非”属于单边运算，即只有一个运算对象，其运算符为一条上横线。规则如下：

$$\overline{1} = 0 \quad \overline{0} = 1 \tag{3.5}$$

【例 3-5】 求数 10011011 的“非”。

解：只要对 10011011 按位取反即可。得：

$$\overline{10011011\text{B}} = 01100100\text{B}$$

表 3-2 给出了这 3 种基本逻辑在开关电路中的表示。

表 3-2 逻辑关系及其电路表述

逻辑关系	运算符	逻辑关系描述	电路表示
逻辑与	∧	当且仅当输入全为 1(真)时，输出才为 1(真)	A B Y
逻辑或	∨	当且仅当输入全为 0(假)时，输出才为 0(假)	A B Y

续表

逻辑关系	运算符	逻辑关系描述	电路表示
逻辑非	上横线	输入为1(真),输出则为0(假),反之亦然	

3.1.3 其他逻辑运算

通过对基本逻辑关系的变换,可以生成其他一些逻辑关系。常见的有“与非”运算、“或非”运算、“异或”运算和“同或”运算等。

1. “与非”运算

“与非”运算是“与”运算和“非”运算的组合,是对“与”运算的结果再求“非”。可以用以下逻辑函数表示:

$$Y = \overline{A \wedge B} \tag{3.6}$$

【例 3-6】 设 A=11011010B,B=10010110B,计算 $Y=\overline{A \wedge B}$。

解:先计算 $A \wedge B$,有

$$\begin{array}{r} 11011010 \\ \wedge\, 10010110 \\ \hline 10010010 \end{array}$$

再对相“与”的结果按位取反,就得到“与非”运算结果:$\overline{10010010}=01101101$。

2. “或非”运算

和“与非”运算类似,“或非”运算是“或”运算和“非” 运算的组合。即在或运算基础上,再对结果求“非”。“或非”运算的逻辑函数表达式为

$$Y = \overline{A \vee B} \tag{3.7}$$

【例 3-7】 设 A=11011001B,B=11111111B,计算 $Y=\overline{A \vee B}$。

解:先计算 $A \vee B$,有

$$\begin{array}{r} 11011001 \\ \vee\, 11111111 \\ \hline 11111111 \end{array}$$

再对结果求“非”,得:$Y=\overline{A \vee B}=\overline{11011001 \vee 11111111}=\overline{11111111}=00000000$。

3. “异或”运算

“异或”逻辑关系是在“与”、“或”、“非”3 种基本逻辑运算基础上的变换。其逻辑代数表达式为

$$Y = \overline{A} \cdot B + A \cdot \overline{B} \tag{3.8}$$

"异或"运算是对两个变量的逻辑运算,用符号⊕表示:

$$Y = A \oplus B$$

【例 3-8】 计算 11010011B⊕10100110B。

解:

$$\begin{array}{r} 11010011 \\ \oplus 10100110 \\ \hline 01110101 \end{array}$$

即 11010011B⊕10100110B=01110101B。

二进制数的"异或"运算可以看作不进位的"按位加",或者不借位的"按位减"。

4. "同或"运算

"同或"运算是在"异或"运算的基础上再进行"非"运算的结果,所以,其运算规则可以直接表示为式(3.9)的函数表达式:

$$Y = \overline{A \oplus B} \tag{3.9}$$

【例 3-9】 设 A=11010011,B=10100110,计算 $Y=\overline{A \oplus B}$。

解: 按照"同或"逻辑关系,先对两数求"异或",再做"非"运算,即对"异或"运算后的结果再按位取反,就得到两数相"同或"的结果:

$$\overline{11010011\text{B} \oplus 10100110\text{B}} = 100010101\text{B}$$

3.2 逻 辑 电 路

实现各种逻辑运算的电路称为逻辑门。本节介绍几种常用的逻辑门,以及由它们组合构成的计算机中的一些基本逻辑电路。作为入门级教材,本书将不涉及逻辑电路的内部结构,仅从应用的角度出发介绍它们的逻辑功能和符号表示。

3.2.1 基本逻辑门

实现上述各种基本逻辑运算的电路称为基本逻辑门电路。逻辑门是由若干半导体元件经过一定的组合,能够实现某一种逻辑关系的集成电路。物理上的一片集成电路芯片上,通常会集成若干个具有同样逻辑关系的逻辑门,例如将 4 个"与"门集成在一片芯片上构成的 4"与"门电路。

1. "与"门(AND gate)

"与"门是对多个逻辑变量(取值范围只有 0 和 1)进行"与"运算的门电路,可以有多位输入,但只有 1 位输出。图 3-1 所示是两输入"与"门的两种逻辑符号。图中,A 和 B 为"与"门的输入,Y 是"与"门的输出。**当且仅当输入全为 1 时,输出为 1;否则输出为 0。**

"与"门输入和输出之间的关系可以表示为式(3.2),也可用表 3-3 来表示(该表也称

为“与”逻辑的真值表)。若从电平变化的角度描述,则仅当“与”门的输入 A 和 B 都是高电平时,输出 Y 才是高电平,否则 Y 就输出低电平。

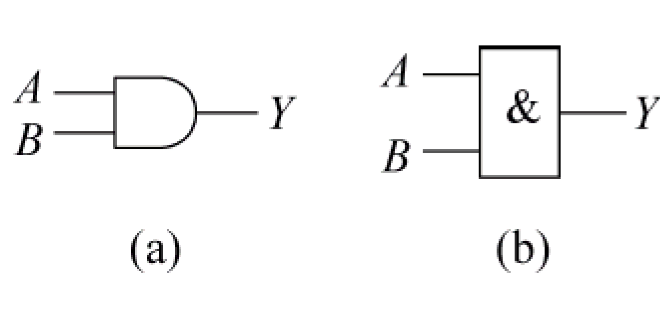
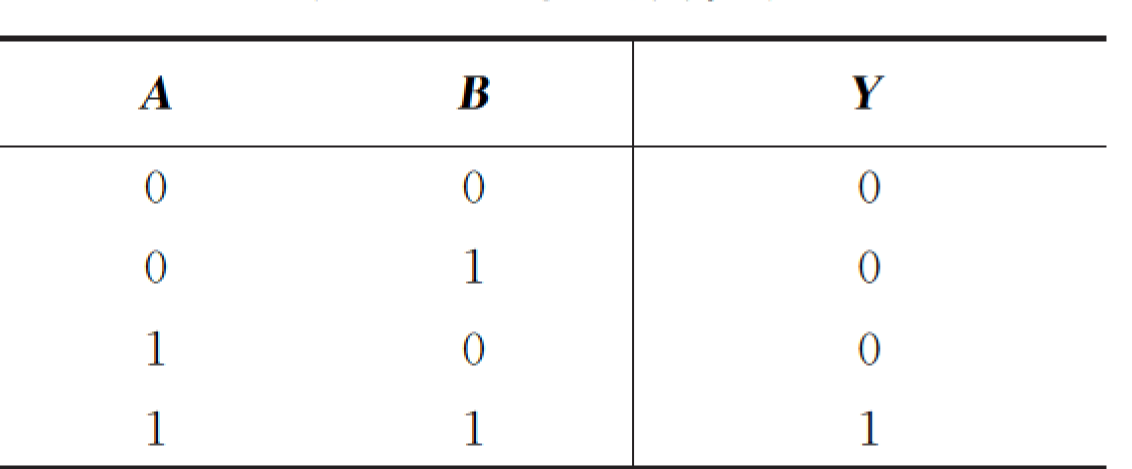


图 3-1 “与”门的逻辑符号

表 3-3 “与”门真值表



A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

对图 3-1,有两点需要注意:

(1) 图中仅画出了 2 位输入(A 和 B),实际的“与”门电路可以有多位输入。

(2) 图中给出了“与”门的两种表示方法,其中,(a)为 IEEE 推荐符号,(b)为中国国家标准规定使用的符号,这两种图符目前均可以使用(以下类同)。为描述上的方便,本书后面以图中的(b)为主。

2. “或”门(OR gate)

“或”门是对多个逻辑变量进行“或”运算的门电路。和“与”门一样,也是多输入、单输出的门电路。其逻辑符号如图 3-2 所示。图中,A、B 为“或”门的输入,Y 是“或”门的输出。**当且仅当输入全为 0 时,输出为 0;输入有一位是 1,输出则为 1**。或者说:输入 A 和 B 只要有一个是高电平,输出 Y 就为高电平;否则 Y 输出低电平。

两输入“或”门可用图 3-2 所示的两种图符之一来表示,其真值表见表 3-4。和“与”门类似,本书中也将更多使用(b)所示的图符。

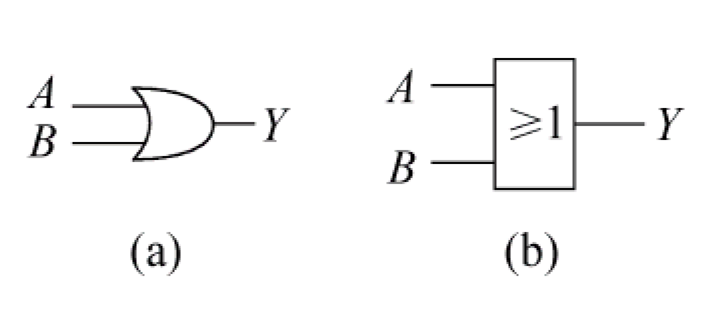


图 3-2 “或”门的逻辑符号

表 3-4 “或”门真值表

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

3. “非”门(NOT gate)

“非”门又称为反相器,是对单一逻辑变量进行“非”运算的门电路(如图 3-3 所示),其输入变量 A 与输出变量 Y 之间的关系可用以下函数式表示:

$$Y = \overline{A} \tag{3.10}$$

“非”运算也称求反运算,变量 A 上的上划线 $\overline{\quad}$ 在数字电路中表示反相之意。表 3-5 为“非”门的逻辑关系真值表。

计算机是由成千上万各种各样的逻辑门电路经过组合构成的,可以说,逻辑门是构成计算机的最小“细胞”单位,但任何复杂的逻辑门都是在基本逻辑门基础上的组合、变换。

因此，基本逻辑门及其逻辑关系是学习计算机科学非常重要的基础。

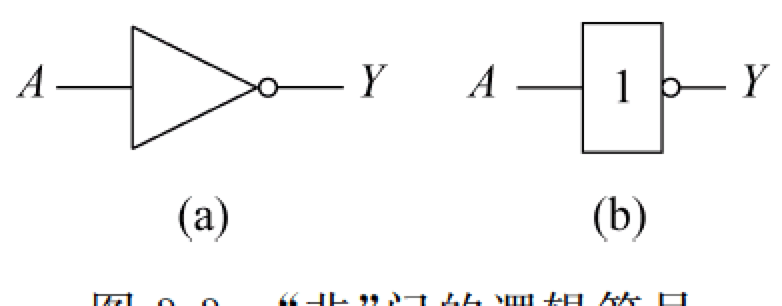


图 3-3 “非”门的逻辑符号

表 3-5 “非”门真值表

A	Y
0	1
1	0

3.2.2 其他常用逻辑门

由基本逻辑门可以组合变换出其他各种逻辑电路。这些逻辑电路中，最常见也是最基本的有“与非”门、“或非”门、“异或”门和“同或”门等。

1. “与非”门(NAND gate)

将“与”门的输出连接到“非”门的输入，就构成了“与非”门。式(3.6)表示的逻辑函数可以表示为图 3-4 所示的两输入“与非”门，图中的小圆圈表示“非”(本书将始终采用(b)图表示方法)。表 3-6 是“与非”门的真值表。

(a) (b)

图 3-4 “与非”门的逻辑符号

表 3-6 “与非”门真值表

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

2. “或非”门(NOR gate)

“或非”门是“或”门和“非”门的组合，即将“或”门的输出连接到“非”门的输入。式 3.7 表示的两变量逻辑函数对应的逻辑门如图 3-5 所示，真值表见表 3-7。

(a) (b)

图 3-5 “或非”门的逻辑符号

表 3-7 “或非”门真值表

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

3. “异或”门(XOR gate)

“异或”门是对两个逻辑变量(请注意这里的表述)进行“异或”运算的门电路，它有 2 位输入，1 位输出，其逻辑符号如图 3-6 所示，真值表见表 3-8。“异或”门输出对输入的关

系可以简单地表述为：输入相同则为 0，输入相异则为 1。即，当两输入状态相同时，输出结果为 0；两输入状态不同时，输出结果为 1。

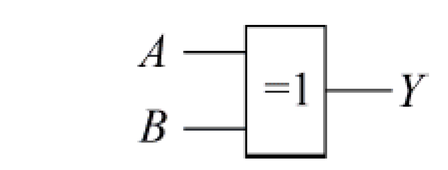
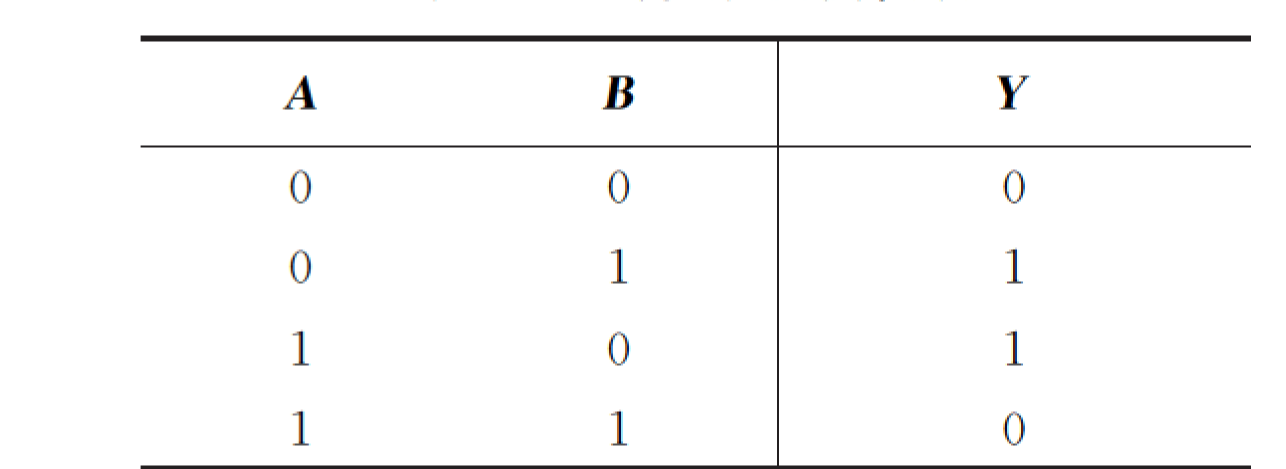


图 3-6 “异或”门的逻辑符号

表 3-8 “异或”门真值表



A	*B*	*Y*
0	0	0
0	1	1
1	0	1
1	1	0

4. “同或”门（XNOR gate）

“同或”门也称“异或非”门，即相当于将“异或”门的输出再连接到“非”门的输入的逻辑门电路，同样有 2 位输入，1 位输出。

“同或”门的逻辑图符如图 3-7 所示，其逻辑关系见表 3-9。“同或”门输出与输入之间的逻辑关系也可以简单表述为：输入相同则为 1，输入相异则为 0。即，当两输入状态相同时，输出结果为 1；两输入状态不同时，输出结果为 0。

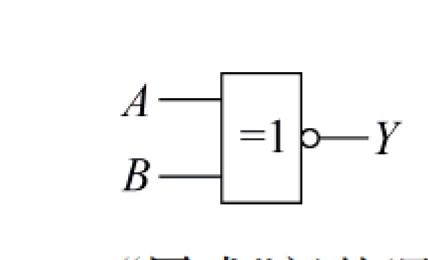


图 3-7 “同或”门的逻辑符号

表 3-9 “同或”门真值表

A	*B*	*Y*
0	0	1
0	1	0
1	0	0
1	1	1

上述 4 种门电路都是由基本逻辑门构造而成的。在实际数字电路设计中，“与非”门、“或非”门和“异或”门的使用非常广泛，“同或”门则相对使用得较少。

3.2.3 触发器

触发器（trigger，flip-flop），也称为双稳态多谐振荡器（bistable multivibrator），是一种可以使其输出端在高电平和低电平两种状态间相互翻转的逻辑电路。即，可以在外部触发信号的作用下，使输出由高电平变为低电平，或由低电平变换为高电平。而当外部触发信号不出现时，只要保持供电，其输出端的状态就可以稳定地保持不变。

触发器的这一特性，使其可以用于存储二进制数 0 和 1，也就是说，它是具有记忆功能的逻辑组件。

1. 触发器电路构造

触发器由各种逻辑门构成，同时，它也是计算机中更复杂电路的基本组成部件。在图 3-8 所示的电路中，两个“与非”门 G_1、G_2 的输入端和输出端分别交叉连接在一起，这就

构成了一种新的功能器件，称为RS触发器。

RS触发器是最基本的一种触发器，有两个输入端 R、S 和两个输出端 Q、$\bar{Q}$。由图3-8可以看出，当 $S=0$，$R=1$ 时，Q 端输出高电平，$\bar{Q}$ 端输出低电平；而当 $S=1$，$R=0$ 时，$\bar{Q}$ 端输出高电平，Q 端输出低电平。即，**当在触发器的两个输入端加上不同逻辑电平时，其输出端 Q 和 $\bar{Q}$ 存在两种互补的稳定状态**。所以，RS触发器属于双稳态电路。

一般情况下，Q 端的状态被视为触发器的输出状态。因为 $S=0$ 会直接使 $Q=1$，所以称 S 端为置1端（或置位端）；当 $R=0$，$S=1$ 时，$Q=0$。所以，R 端也称为触发器的复位端。

当 $R=S=1$ 时，“与非”门的输出不受影响，触发器状态保持不变；当需要触发器翻转时，要求在某一个输入端加上负脉冲。R 和 S 端不允许同时为低电平。

RS触发器的输入输出逻辑关系如表3-10所示。

表3-10　RS触发器逻辑真值表

R	S	Q	$\bar{Q}$
0	0	—	—
0	1	0	1
1	0	1	0
1	1	×	×

虽然RS触发器是由两个“与非”门组成的，但它具有独立的逻辑功能，是一种可以独立存在的逻辑器件。因此，RS触发器可以抽象为图3-9所示的逻辑符号（图中，S 和 R 端的小圆圈表示是低电平有效）。

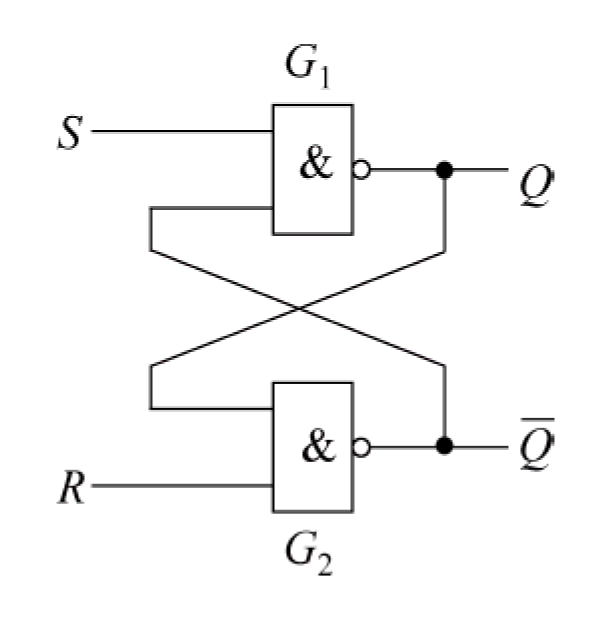


图3-8　RS触发器逻辑电路

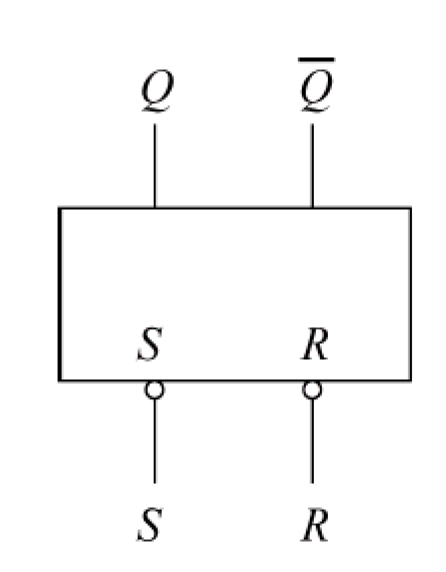


图3-9　RS触发器逻辑符号

图3-10所示的是另一种常见的触发器的基本原理图（没有考虑置位端和复位端），称为D触发器。它是在RS触发器的基础上又加入了两个“与非”门。其中，D 为输入端，CP为控制端。由图可知，当CP=0时，输入端 D 被封锁（**当与门或者与非门的输入端有一位为低电平时，其输出状态就被确定，此时其他输入端的状态将对输出不再产生影响，故称为封锁**），输出保持不变；当CP=1时，输出 Q 将会随着 D 端状态的变换而变换。

作为一种有独立功能的逻辑器件，D触发器也同样可以抽象为图3-11所示的逻辑符号。

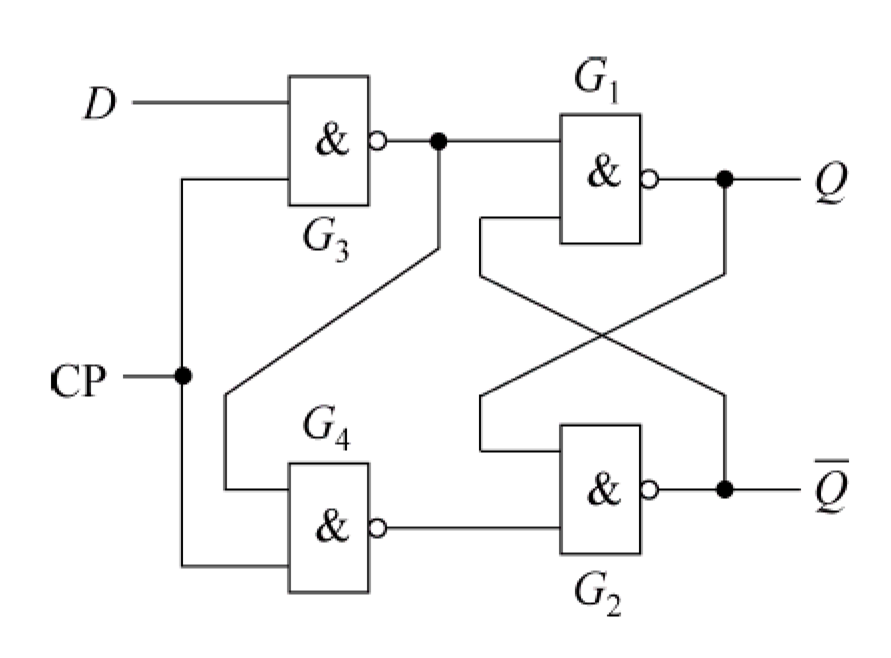


图3-10　D触发器基本原理图

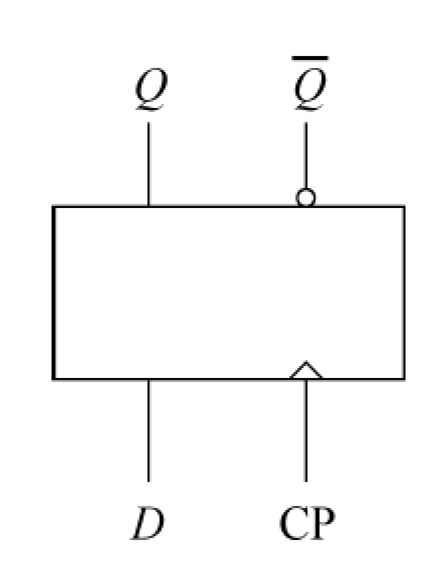


图3-11　D触发器逻辑符号

思考 读者是否可尝试自己总结一下 D 触发器的逻辑真值表?

2. 由触发器引起的思考

触发器是具有记忆功能的器件(也说它具有对信息的保持能力、保存能力或锁存能力)。一个触发器能够存储 1 位二进制码,N 个触发器就可以存储 N 位二进制数(还记得一个内存单元是多少位吗? 可以想一下至少需要多少个触发器)。触发器的这种特性,使它成为构成计算机存储装置(寄存器、存储器等)的基本单元。

从基本逻辑门到触发器,从简单的 RS 触发器到稍微复杂一点的 D 触发器,从基本的 D 触发器再到图 3-12 所示的考虑了置位和复位信号的 D 触发器。可以看出,虽然电路的复杂度在逐步加深,但从本质上,它们都是基本逻辑门的组合。每用若干个基本逻辑门组成一个具有独立功能的部件后,这个部件就可以**封装**为一个整体,用一个逻辑符号来表示(如上述的 RS 触发器和 D 触发器)。此时,就可以不需要再了解它们内部的构造细节,而只要了解它们的整体功能和输入输出间的逻辑关系就可以了。

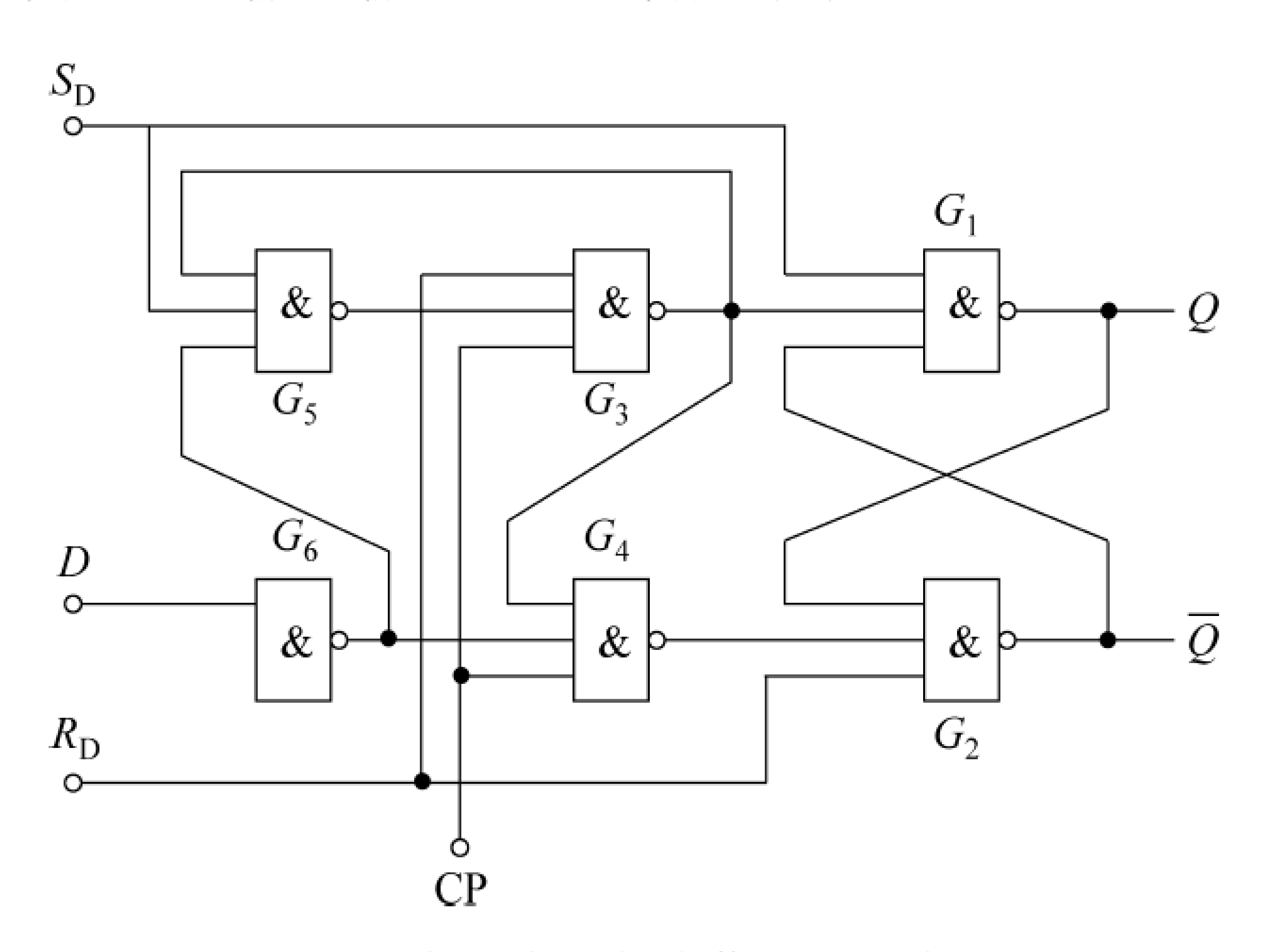


图 3-12 带置位和复位信号的 D 触发器

进一步,用触发器可以构造出计算机中的寄存器或存储器。图 3-13 是由 4 个 D 触发器构成的 4 位移位寄存器。这里,D 触发器成为寄存器的基础构件。如果将寄存器作为一个整体考虑(事实上,寄存器本身在逻辑上就是一个独立部件),那就可以不再关心它内部有几个触发器以及它们之间的连接关系,而只需要了解寄存器本身的功能和工作原理。

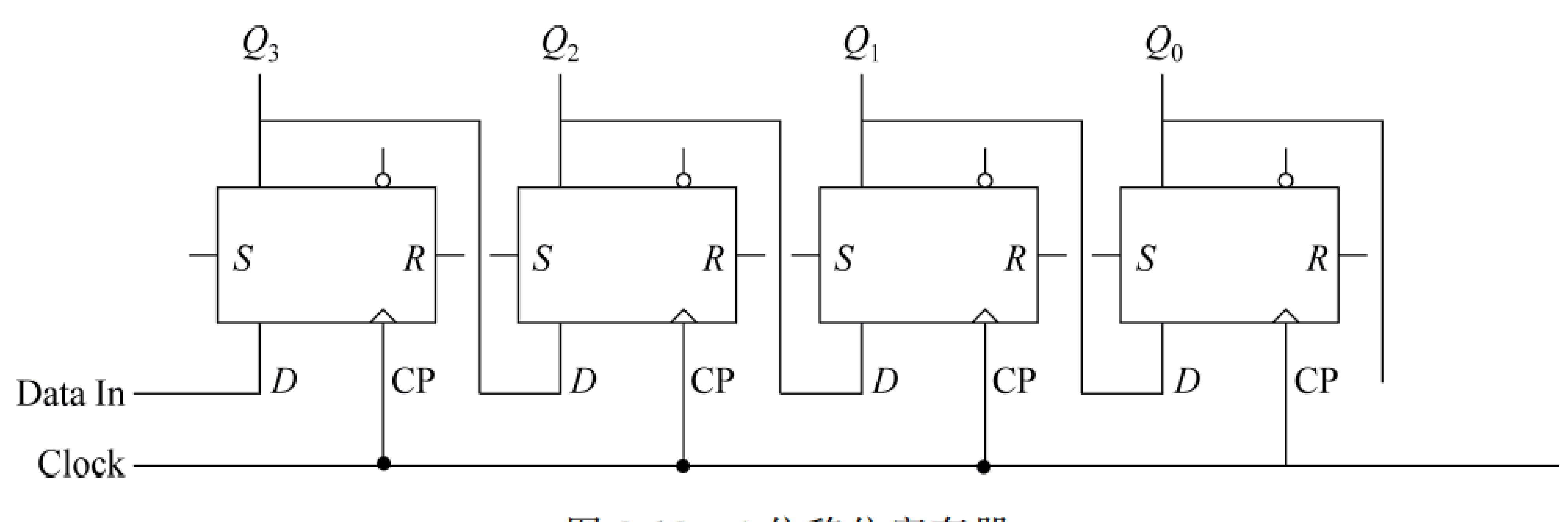


图 3-13 4 位移位寄存器

更进一步,寄存器、存储器这样的存储装置都是计算机硬件系统中的部件之一,由第1章的描述已知,硬件系统中除了存储装置,还有控制逻辑部件、运算器、I/O接口等。如果将计算机作为一个整体(计算机也的确就是一个独立、完整的机器),那么它内部各种部件之间的连接关系也就可以被封装了。

由此可以看出,计算机硬件系统实际上就是以基本逻辑门(当然还要加上一些其他辅助电路)为基础,经过一层一层地组合、封装而构成。整个系统可以分为若干层次,每一层都包含若干低一层的器件(或说由若干低层器件构成),并且用一个"符号"①实现对低一层的封装。这个过程称为硬件结构上的**抽象**。可以用图3-14来示意②。事实上,计算机硬件系统就是由若干基本逻辑部件经过一层一层的抽象构成的。

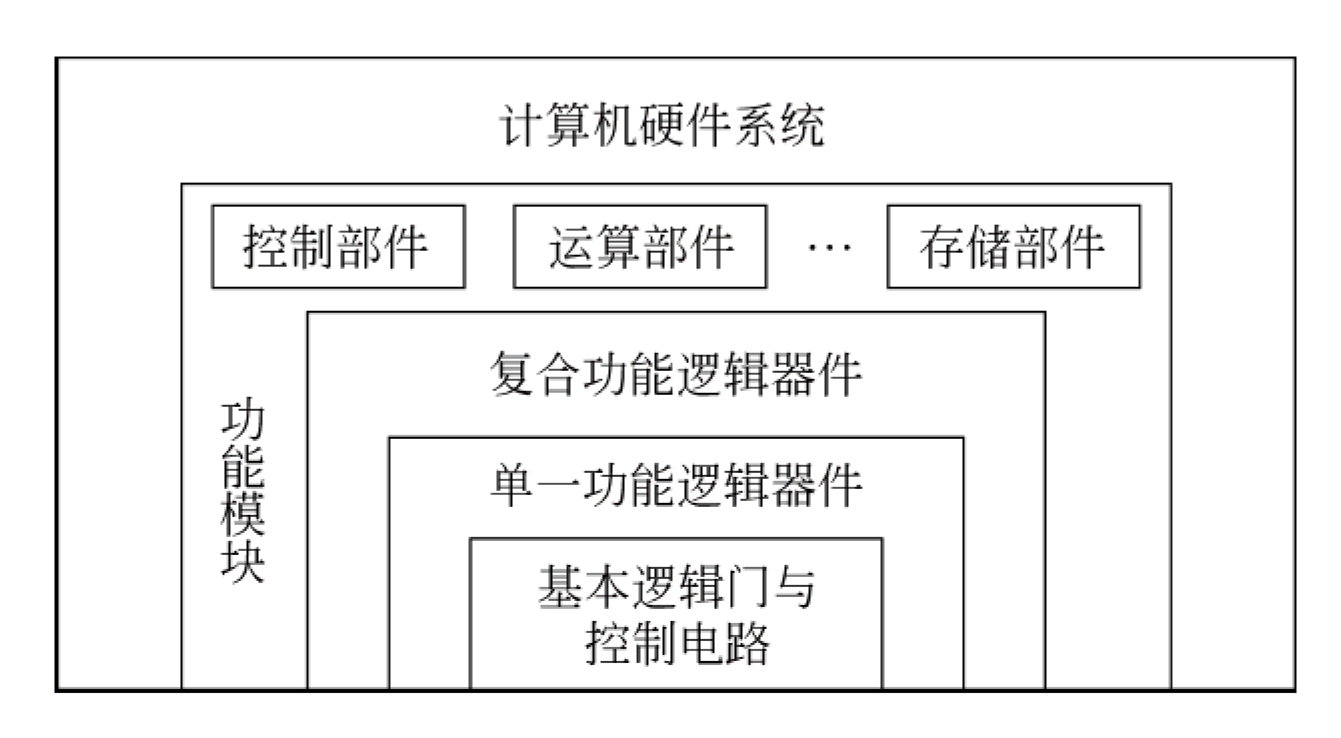


图3-14　硬件系统的分层抽象示意图

3.2.4　加法器

根据逻辑功能的不同特点,数字电路通常可以分为组合逻辑电路和时序逻辑电路两大类。**组合逻辑电路**在逻辑功能上的特点是任意时刻的输出仅仅取决于该时刻的输入,与电路原来的状态无关。如各种门电路等。而**时序逻辑电路**是指电路任何时刻的稳态输出不仅取决于当前的输入,还与前一时刻输入形成的状态有关,即具有记忆功能。触发器就属于时序逻辑电路,而加法器则属于组合逻辑电路。

计算机能够实现二进制的算术运算,算术运算在计算机中是由CPU的运算器完成的,而运算器的核心部件是算术逻辑单元(ALU)。由第2章的描述已知,减法运算可以通过引入补码而转换为加法运算;乘法运算相当于加法和移位操作;除法是乘法的逆运算,相当于减法和移位操作,而减法又可以转换为加法运算。因此,计算机中的算术运算主要是利用加法来实现的。

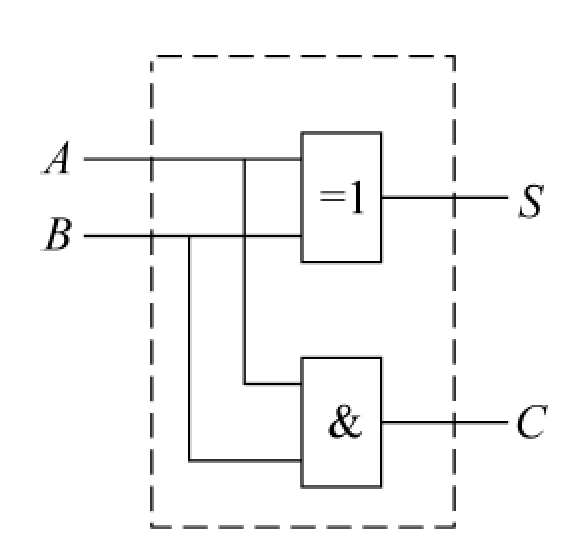
加法器(adder)是一种用于执行加法运算的数字电路,是构成CPU中算术逻辑单元的基础。加法器又分为半加器和全加器。

① 这里的"符号"是一个抽象含义,它既可以是一个具体的逻辑符号(如D触发器),也可以表示一种概念,如文中提到"运算部件",只是一类功能部件的总称,而没有一个具体的符号与它对应。

② 图3-14仅为说明硬件系统的分层"抽象"而设计的示意图,并未包含各种辅助控制电路,所以并不是严谨的硬件系统结构图。

半加器的功能是将两个1位二进制数相加。它具有两个输入(加数、被加数)和两个输出(和、进位)。输出的进位信号代表了输入两个数相加后向高位的进位值。半加器是不考虑来自低位进位的加法器,图3-15给出了半加器的逻辑电路和真值表。它由一个“异或”门和一个“与”门组成,A、B为输入,S为输出的两数之和(sum),C是进位(carry)。由图知:

$$S=A\oplus B,\quad C=A\wedge B$$



输入		输出	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

图3-15　半加器逻辑电路及真值表

如果在半加器中添加一个“或”门来接收低位的进位输出信号,则两个半加器就构成一个全加器,如图3-16所示。

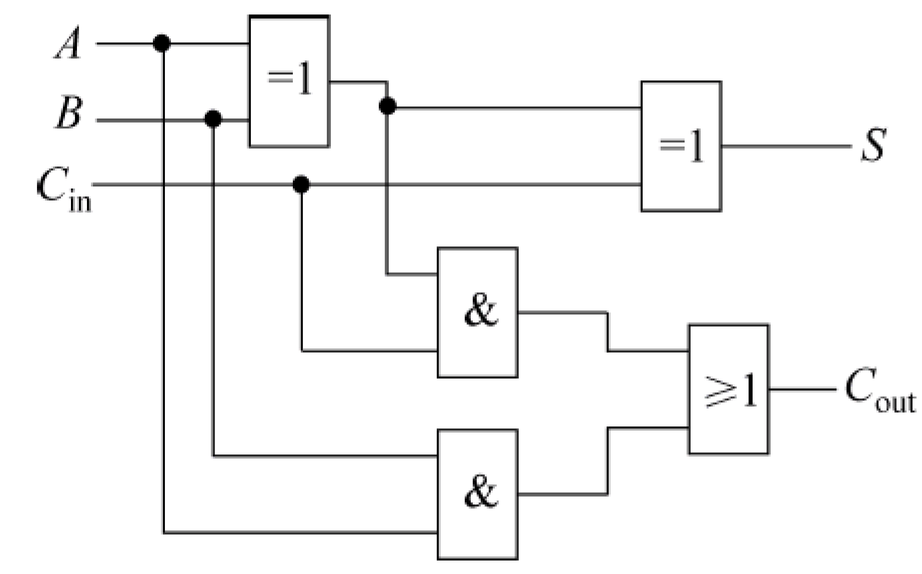
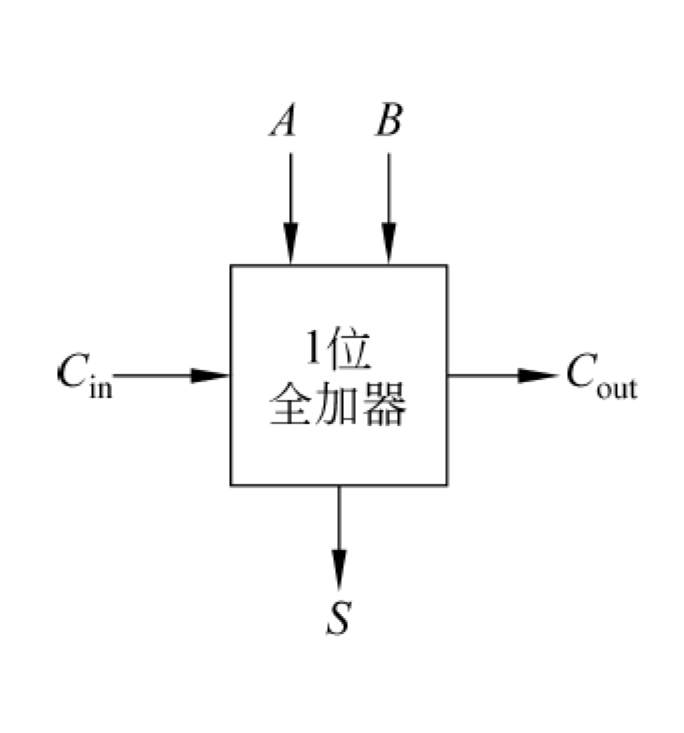


图3-16　1位全加器逻辑电路图

全加器是要考虑进位的加法器。它将两个1位二进制数相加,并根据接收到的低位进位信号,输出相加的和以及向高位的进位。按照分层抽象的方法,图3-16所示的逻辑电路可以用图3-17所示的逻辑符号表示。其中:A和B是分别加数、被加数,C_{in}是来自低位的进位信号,C_{out}是向高位输出的进位信号,S是相加的和。

全加器只能实现1位二进制的加法,因为CPU的字长从早期的8位、16位到现在的32位或64位,都不是只完成1位二进制运算,而需要同时进行多位二进制数的运算。因此,需要在全加器的基础上构造多位加法器。



输入			输出	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

图3-17　1位全加器逻辑符号及真值表

多位加法器主要有涟波进位加法器(ripple-carry adder)和超前进位加法器(carry-lookahead adder)两种。

涟波进位加法器的实现方法比较简单，它用 N 个全加器构成，其中对应低位的全加器将其进位输出信号 C_{out} 连接到高一位全加器的进位输入端 C_{in}，并依次像“涟波”一样将进位信号向前传递。图 3-18 所示为 4 个 1 位全加器构成的 4 位涟波进位加法器。

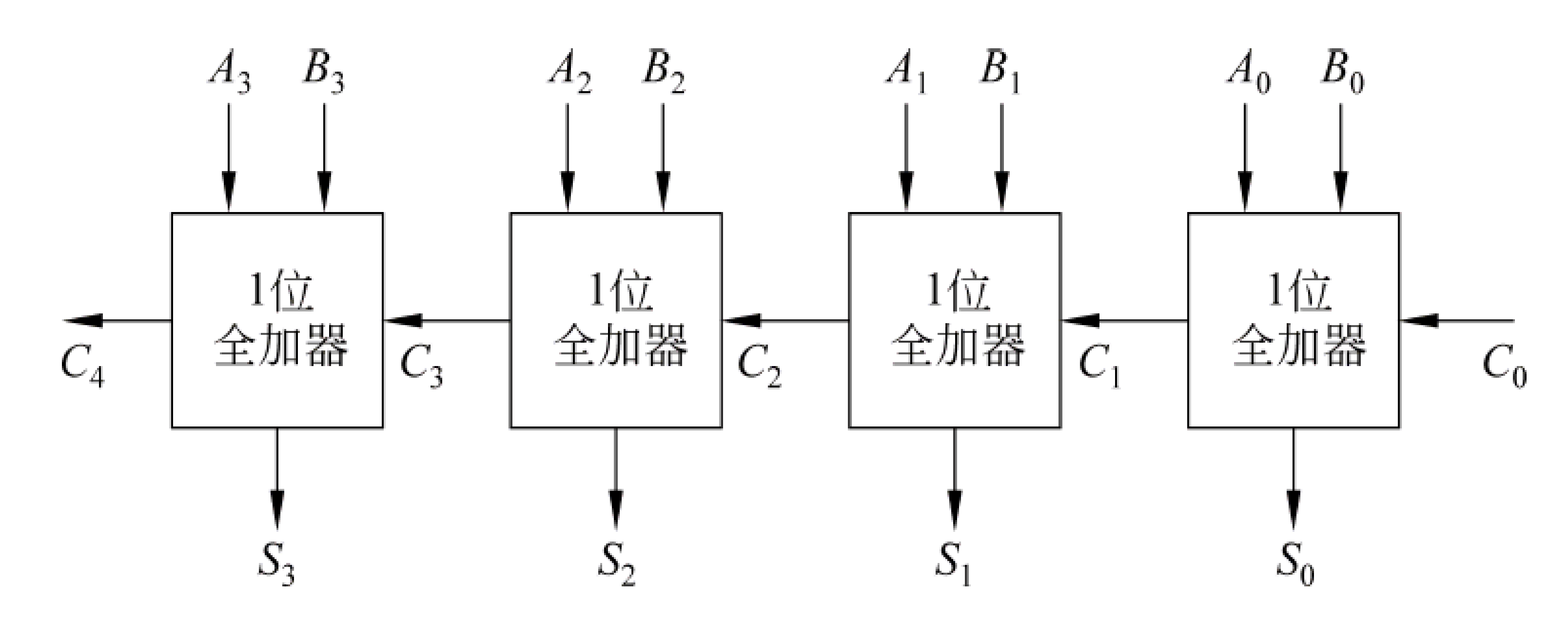


图 3-18　4 位涟波进位加法器

进位信号向前传递会产生传递延迟，最低位向前的进位需要顺序通过所有全加器才能产生最终的结果，这是这种加法器的主要缺点。

目前普遍使用的并行加法器是超前进位加法器。因篇幅原因，不再做进一步描述。

在这里介绍加法器，并非为了使读者学习加法器的设计（这样粗浅的介绍远达不到设计的可能），主要的目的有两个：一是帮助读者了解计算机进行加法运算的基本原理，更重要的是希望读者能通过本节由基本逻辑门到一些常见逻辑电路再到加法器（也可以说是计算机核心部件之一）的逐层抽象，初步建立硬件系统的构造思维模式。

3.3　冯·诺依曼结构

计算机历经半个多世纪的发展，在运算速度、存储能力及整体功能上都有了巨大的进步，根据摩尔定律①，计算机的整体性能每两年就会翻一番，事实也的确如此。今天，一台普通个人计算机的性能已远远超过 50 年前的巨型计算机。虽然发生了如此大的进步，但如今微型计算机的体系结构和工作原理依然沿用着冯·诺依曼 50 多年前的设计思想。

3.3.1　程序和指令

现代计算机不仅能够进行各种复杂的数值计算，还能够模拟人类的思维分析和处理各种事物。那么，计算机为什么能够做这么多的事情？它是如何完成每一项任务的？

计算机之所以能够按照要求完成一项一项的工作，是因为人向它发出了一系列的命令，这些命令通过输入设备以一定的方式送入计算机，并且能够为计算机所识别。这种能够被计算机识别的命令称为指令，一台计算机能够识别的所有指令的集合称为该计算机的指令系统，而保证对指令的这种执行能力的是计算机的硬件系统。

当人们需要计算机完成某项任务的时候，首先要将任务分解为若干个基本操作的集

① 当价格不变时，集成电路上可容纳的晶体管数目约每隔 18 个月便会增加一倍，性能也将提升一倍。

合，并将每一种操作转换为相应的指令，按一定的顺序组织起来，这就是程序。计算机完成的任何任务都是通过执行程序完成的。例如，在需要解一道数学题时，要先把题目的解算步骤按照一定的顺序用计算机能够识别的指令书写出来，命令计算机执行规定的操作。这些指令的序列就组成了程序，如图 3-19 所示。

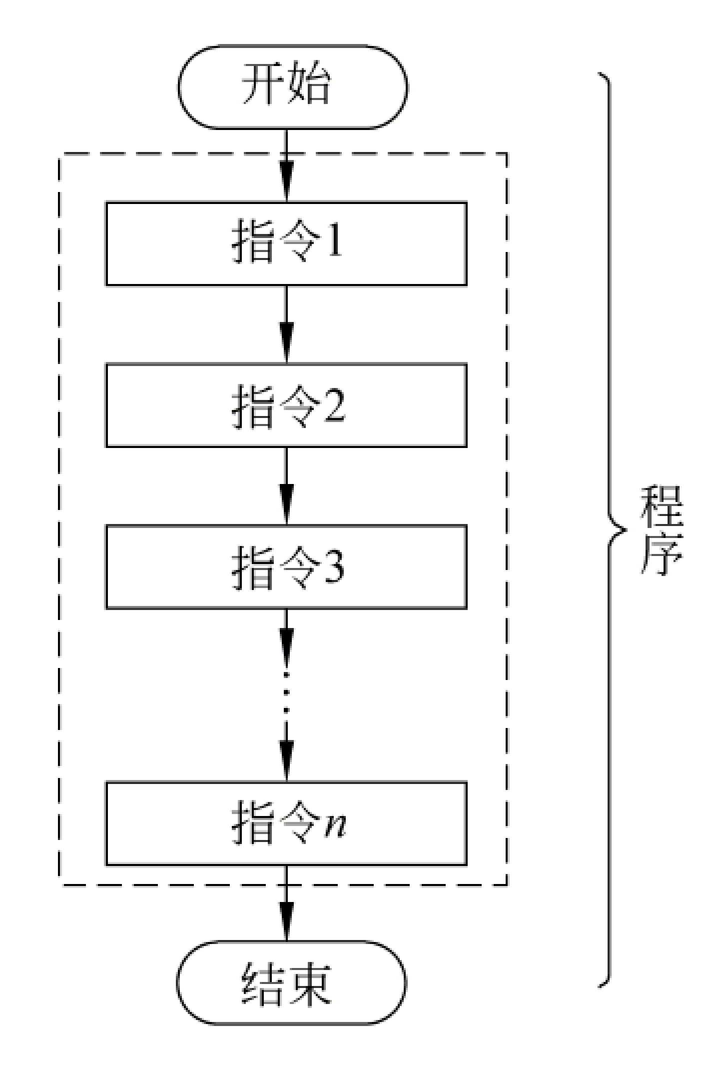


图 3-19 程序中的指令

计算机硬件能够直接识别并执行的指令称为机器指令。它们全部由 0 和 1 这样的二进制编码组成，其操作通过硬件逻辑电路实现。

不同的计算机系统通常都具有自己特有的指令系统，其指令在格式上也会有一些区别，但一般都包含这样 3 种信息，即完成何种操作（操作性质，如加、减、乘、除等）、对谁操作（操作的对象）以及操作结果的存放处。表征指令操作性质或者功能的称为操作码，表征操作对象的称为操作数（或地址码[①]）。指令的一般格式如图 3-20 所示。

操作码(OPC)	操作数的目标地址，操作数的源地址

图 3-20 指令的一般格式

每台计算机都拥有由各种类型的机器指令组成的指令系统。指令系统的功能是否强大，指令类型是否丰富，决定了计算机的能力，也影响着计算机的结构。指令的不同组合方式可以构成完成不同任务的程序。计算机严格按照程序安排的指令顺序，有条不紊地执行规定的操作，完成预定任务。因此，程序是实现既定任务的指令序列，其中的每条指令都表示计算机执行的一项基本操作。一台计算机的指令种类是有限的，但通过人们的精心设计，可编写出无限多个实现各种任务处理的程序。

3.3.2 冯·诺依曼计算机基本结构

冯·诺依曼计算机的典型结构模型如图 3-21 所示，具有一个存储器、一个控制器、一个运算器以及输入和输出设备。所有的输入和输出都需要通过运算器。人们将这种结构称为冯·诺依曼结构，也称普林斯顿结构（Princeton architecture）。

冯·诺依曼结构的核心设计思想主要体现在：存储程序控制原理，以运算器为核心，采用二进制。可进一步描述为：

（1）将计算过程描述为由许多条指令按一定顺序组成的程序，并放入存储器保存。

（2）程序中的指令和数据都采用二进制编码（抛弃了十进制记数的设计思路），且能够被执行该程序的计算机所识别。

（3）指令和数据可一起存放在存储器中，并作同样处理。

（4）指令按其在存储器中存放的顺序执行，存储器的字长固定并按顺序线性编址。

---

① 表示运算数据及运算结果的存放处。

（5）由控制器控制整个程序和数据的存取以及程序的执行。

（6）计算机由运算器、逻辑控制装置、存储器、输入设备和输出设备5个部分组成，以运算器为核心，所有的执行都经过运算器。

冯·诺依曼计算机的设计思想简化了计算机的结构，大大提高了计算机的工作速度。图3-21是冯·诺依曼计算机的结构示意图。

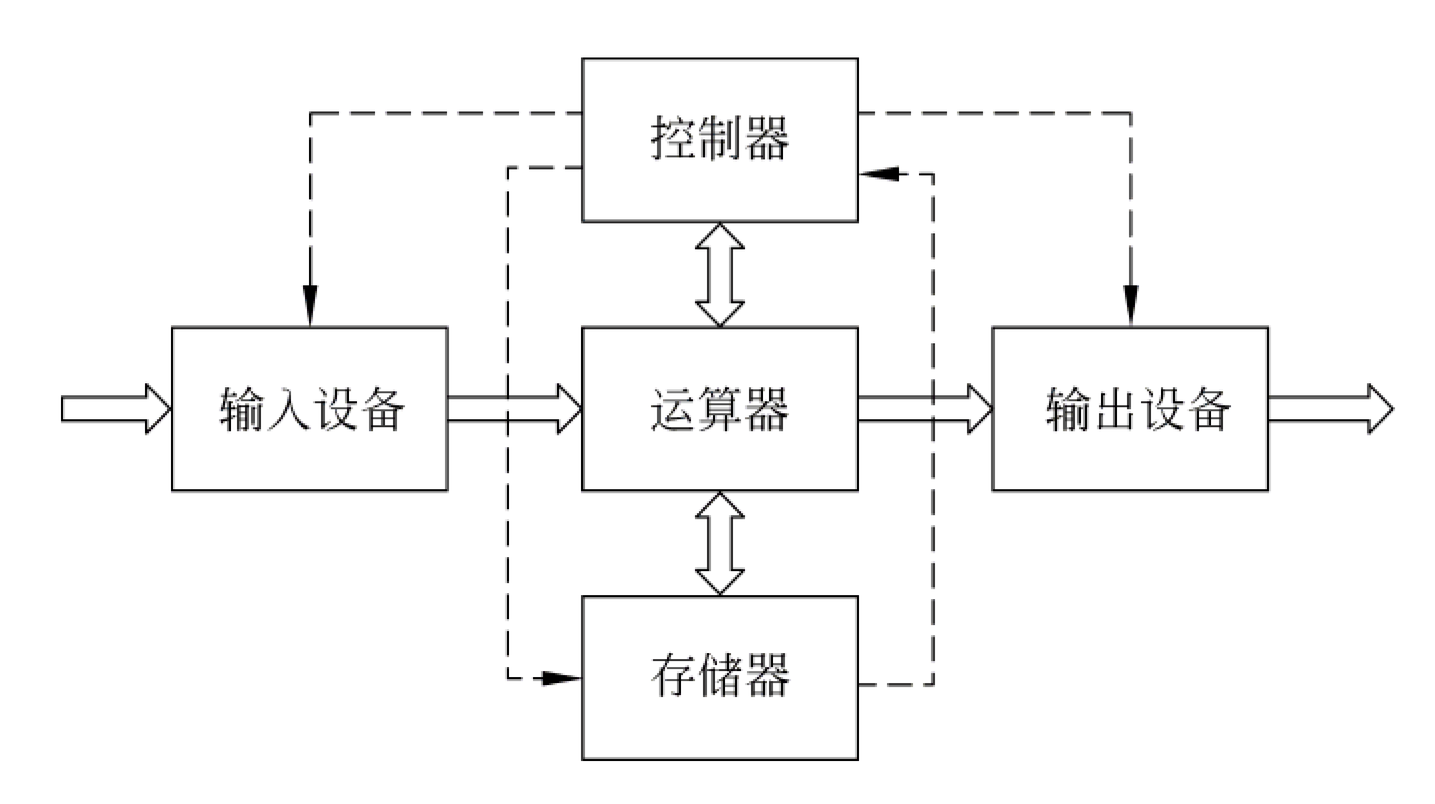


图3-21　冯·诺依曼计算机结构示意图

半个多世纪过去了，虽然计算机软硬件技术都有了飞速的发展，但直至今天，微型计算机的基本结构形式并没有明显的突破，仍属于冯·诺依曼结构。计算机的基本工作原理仍然是存储程序控制原理。当然，二进制也依然是计算机硬件唯一能够直接识别的数制。

3.4　冯·诺依曼计算机基本原理

计算机的工作过程就是执行程序的过程，而程序是指令的序列。所以，计算机的工作过程就是一条条执行指令的过程。

3.4.1　指令的执行过程

由3.3.1节可知，指令是控制计算机完成某种操作并能够被计算机硬件所识别的命令。因此，指令才有如图3-20所示的格式（即包含指令码和操作数）。根据冯·诺依曼结构原理，程序在被执行前先要存放在（内）存储器中（为什么？学完3.6节就应该清楚了），而程序的执行需要由CPU完成。因此，计算机在执行程序时，首先需要按某种顺序将指令从内存储器中取出（一次读取一条指令）并送入处理器，处理器分析指令要完成的动作，明确其操作性质和操作的对象，再去存储器中读取相应的操作数（如果需要），然后执行相应的操作，最后将运算结果存放到内存储器中（如果这个结果不需要送到内存，当然就可以不送了）。这一过程直到遇到结束程序运行的指令才停止。

因此，指令的执行过程可简单地描述为5个基本步骤：取指令、分析指令、读取操作数、执行指令和存放结果。图3-22给出了一条指令的执行流程。

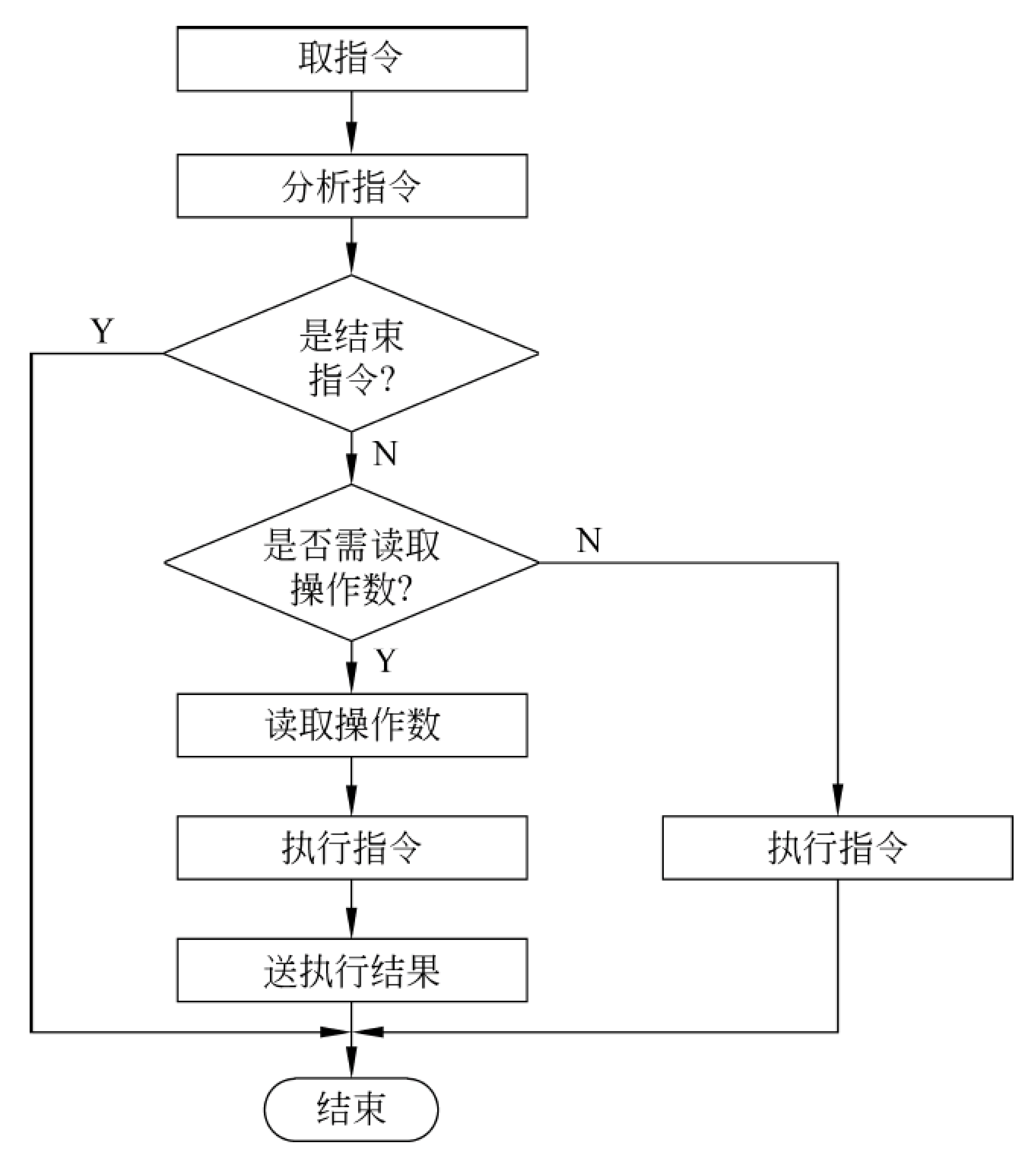


图 3-22　指令的执行过程

图 3-22 中的“是否需读取操作数”的分支，表示不是每一条指令都需要到内存中去读取操作数。当然，这不表示指令没有操作的对象，而是操作的对象可能是处理器本身。

以下暂且只讨论仅包括取指令、分析指令（也称指令译码）和执行指令这 3 个基本步骤时指令的执行方式。

在现代微处理器中，取指令、分析指令和执行指令的工作是由 3 个部件分别完成的。这 3 个部件可以同时工作（并行工作），也可以顺序方式工作（串行工作）。

1. 顺序工作方式

所谓顺序工作方式是指取指令、分析指令和执行 3 个部件依次工作，前一个部件工作结束后，下一个部件才开始工作。

指令顺序工作方式的工作过程如图 3-23 所示。在早期计算机系统中均采用这样的执行方式。

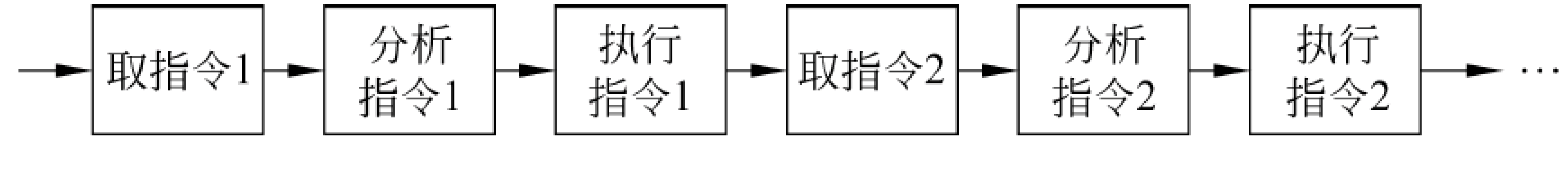


图 3-23　指令顺序执行方式示意图

顺序工作方式的优点是控制系统简单，实现比较容易；另外也节省硬件设备，使成本较低。缺点主要有两个，一是微处理器执行指令的速度比较慢，因为只有在上一条指令执行结束后，才能够执行下一条指令；二是处理器内部各个功能部件的利用率较低。如果以图 3-23 所示的流程工作，则在取指令部件从内存中读取指令时，分析指令和执行指令部

件都处于空闲状态；同样，在指令执行时也不能同时去取指令或分析指令。因此，顺序执行方式时系统总的效率是比较低的，各功能部件不能充分发挥作用。采用顺序方式执行 n 条指令所用时间可用式(3.11)表示：

$$T_0 = \sum_{i=1}^{n}(t_{取指令i} + t_{分析指令i} + t_{执行指令i}) \tag{3.11}$$

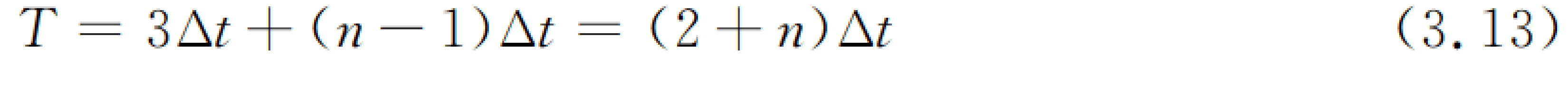
假设计算机取指令、分析指令和执行指令所用的时间相等，均为 Δt，则完成一条指令的时间就是 $3\Delta t$，而执行完 n 条指令需要的时间为

$$T_0 = 3n\Delta t \tag{3.12}$$

2. 并行工作方式

并行工作方式是使上述 3 个功能部件同时工作，即在指令被取入到处理器，开始进行分析的时候，取指令部件就可以去取下一条指令；而当指令分析结束开始被执行时，指令分析部件就可以进行下一条指令的译码工作，同时取指令部件又可以再去取新的指令……这样依次进行，在进入稳定状态后，就可以实现多条指令的并行处理。

图 3-24 给出了并行工作方式下的指令执行过程示意图。图中，当第 1 条指令进入指令分析部件时，取指令部件就开始从内存中取第 2 条指令，假如这 3 个功能部件的执行时间完全相等，均为 Δt，执行第 1 条指令需要的时间为 $3\Delta t$，之后每过一个 Δt 时间，就有一条指令执行完成，则执行 n 条指令所需要的时间为



$$T = 3\Delta t + (n-1)\Delta t = (2+n)\Delta t \tag{3.13}$$

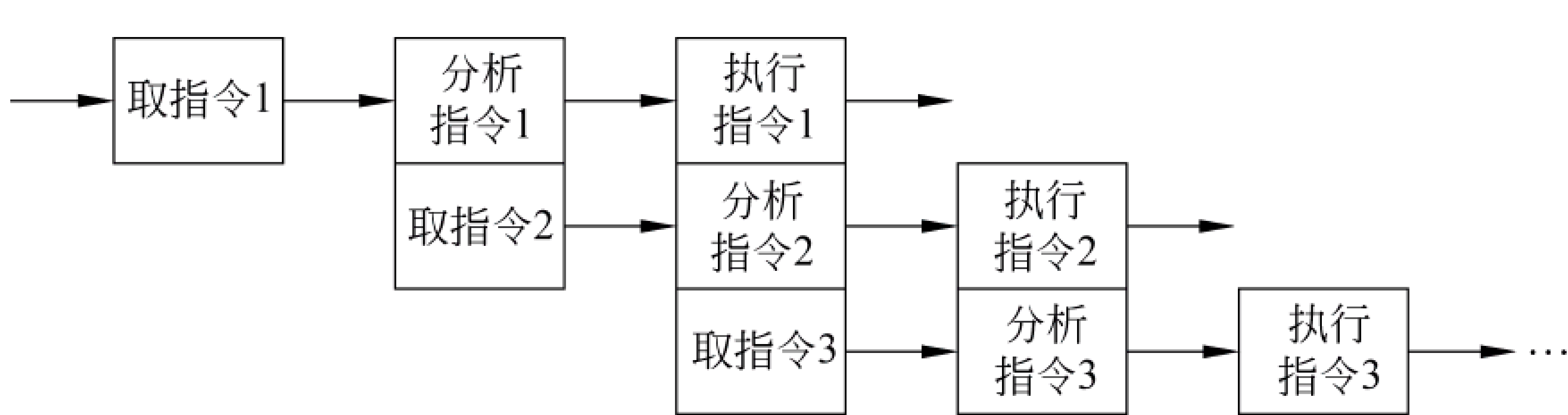


图 3-24　指令并行执行方式示意

由式(3.13)可以看出，与采用顺序执行方式所用的时间 T_0 相比，并行执行方式缩短了系统执行程序的时间，且这种时间上的收益率会随着指令数量 n 的增加而更加显著。

相对于顺序执行方式，并行方式减少的时间量可用系统加速比 S 来描述：

$$S = T_0/T = 3n\Delta t/[(2+n)\Delta t] = 3n/(2+n) \tag{3.14}$$

【例 3-10】 某程序段经编译后生成 10 000 条机器指令，假设取指令、分析指令和执行指令所用的时间均为 t。分别求出使用顺序执行方式和并行流水线方式完成该程序段所需的时间，并说明使用顺序执行方式比并行方式慢多少(即系统的加速比)。

由题目知，n=10 000，$\Delta t = t$，则由式(3.2)，顺序执行完该程序所需时间为

$$T_0 = 3nt = 30\ 000t$$

采用并行流水方式执行该程序需要的时间为

$$T = (2+n)t = 10\ 002t$$

顺序执行与并行方式所耗费时间的比为

$$S = T_0/T = 30\ 000t/10\ 002t = 30\ 000/10\ 002 \approx 3$$

可见,顺序执行方式所花费的时间约为并行方式的 3 倍。

图 3-24 所示的模型是现代计算机流水线控制技术的基本模型。该模型所给出的是理想的情况,即每个部件的工作时间完全相同,也仅在这样的假设下,所示模型的流水线才不会“断流”。这在实际的系统中是不可能的。

为了解决流水线的断流问题,在现代计算机系统中,在取指令和指令译码部分,都设置有指令和数据缓冲栈,可以实现指令和数据的预取和缓存。指令执行部分设置有独立的定点算术逻辑运算部件、浮点运算部件等。另外,加入了预测、分析、多级指令流水线等多项技术,实现对指令和数据的预取和分析,以尽可能地保证流水线的连续。

3.4.2 微型计算机的一般工作过程

计算机的工作过程就是执行程序的过程,也就是逐条执行指令序列的过程。由于每一条指令的执行都包括取指令(含指令译码)和执行指令两个基本阶段,所以,微机的工作过程也就是不断地取指令和执行指令的过程。

当需要计算机完成某项任务时,最基本的工作是首先要使用某一种程序设计语言[①]编写出相应的程序。编写完成后,需要以文件形式(要起个名字)存放在外存储器中,运行时在操作系统控制下通过接口输入到内存。

CPU 是整个计算机的核心,所有程序的执行和控制都是由 CPU 完成的。为了说明微型机的工作过程,在图 1-5 所示的 CPU 基本结构的基础上,给出稍微详尽一点的 CPU 结构模型(如图 3-25 所示)[②]。在该图中,运算器部分中的核心部件就是 ALU。另一个需要关注的部件是程序计数器(Program Counter,PC),它是 CPU 控制程序走向(就是执行完一条指令后下边该执行哪条指令)的“指挥棒”。

进入内存后的程序,会按照逻辑上的顺序依次放入内存各单元。假设程序已存放到内存,当计算机要从停机状态进入运行状态时,处理器内部的程序计数器(PC)会指向程序的第一条指令。当 PC 所指向的指令被取出后,处理器将自行修改 PC 的值,使其指向下一条指令。指令的执行结果会暂存在内存中,最后在操作系统控制下存入外存或由输出设备送出。图 3-26 给出了程序在进入内存后、计算机按顺序执行方式执行一条指令的工作过程:

(1) 控制器将要读取的指令在内存中的地址赋给 PC(图中假设为 04H),并送到地址寄存器 AR。

(2) PC 自动加 1,AR 的内容不变。

(3) 将地址寄存器 AR 的内容发送到地址总线上,并送到内存储器,经地址译码器译码,选中相应的内存单元。

(4) CPU 的控制器发出“读”控制信号。

---

① 关于程序设计语言的相关介绍请参阅本书第 5 章。

② 本书引入该图的目的仅为下面描述的方便。有关 CPU 具体的工作原理请参阅其他硬件系统类书籍。

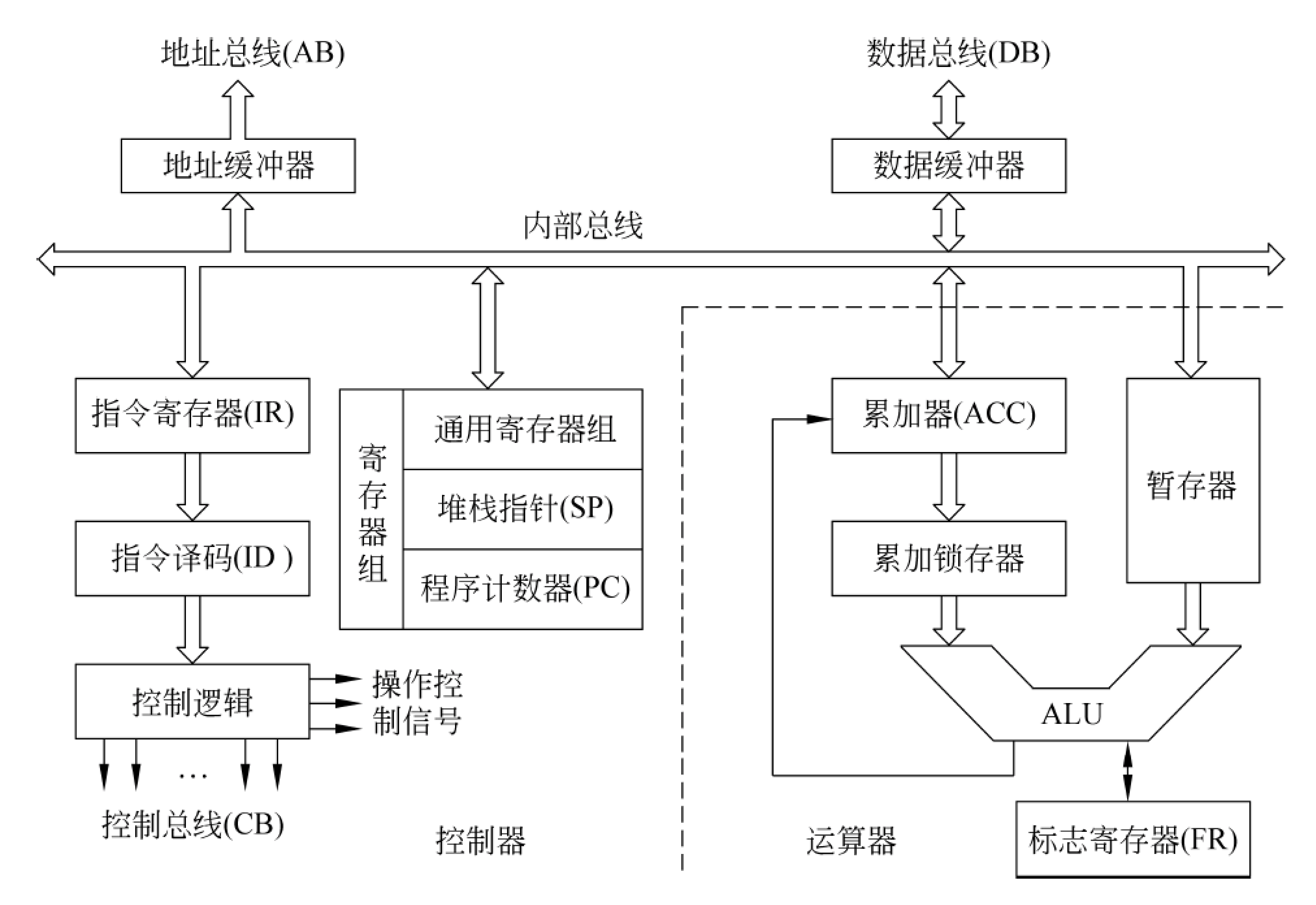


图 3-25　CPU 结构示意图

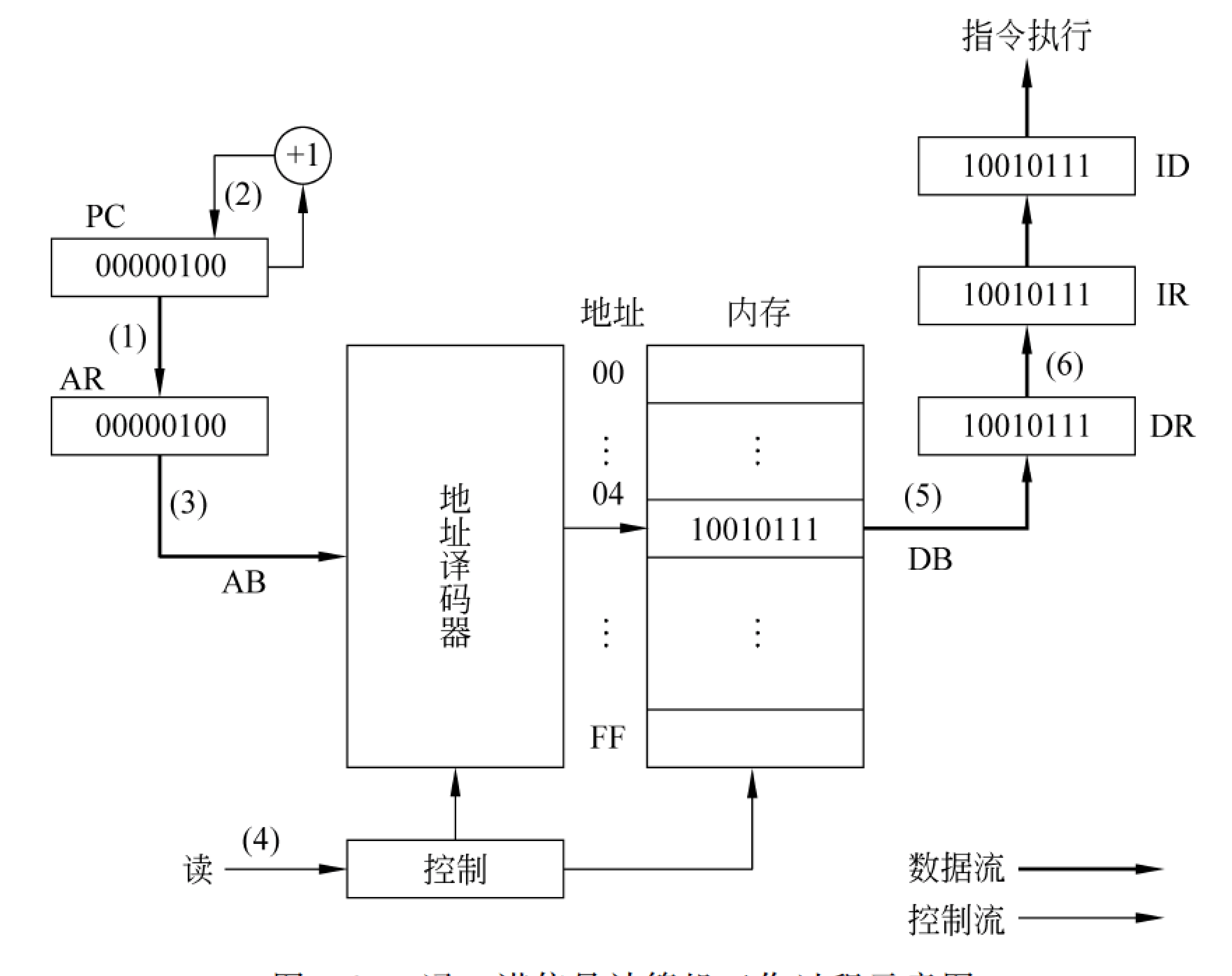


图 3-26　冯・诺依曼计算机工作过程示意图

(5) 在读命令控制下，所选中的内存 04H 号单元中的内容(即指令码，图中假设为 97H，即 10010111)被读出送到数据总线上，并送入数据寄存器 DR。

(6) DR 将读出的指令码送到指令寄存器 IR，然后送指令译码器 ID，进行指令分析。

至此，就完成了一条指令的读取。读取的指令经译码后，若需要再到内存中读取操作数，则继续下述过程：

(7) 发送运算所需操作数的地址。

(8) 读取操作数。

(9) 使运算器开始执行指令。

(10) 发送保存运算结果的地址。

(11) 将运算结果暂存在内存中。

一条指令执行结束后,就转入了下一条指令的取指令阶段。如此周而复始地循环,直到程序中遇到暂停指令方才结束。

上述整个工作过程中,控制器将会发出相应的各种控制信号(如“读”信号、“写”信号等),协调和控制各部件的运行。

应当指出,读操作完成后,04H 单元中的内容 97H 仍保持不变,这种特点称为非破坏性读出(non destructive read out)。这一特点很重要,因为它允许多次从某个存储单元读出同一内容。

处理器向内存中写入执行结果的过程与“读”操作过程类似,不同的是:此时控制器发出的是“写”命令。CPU 将要写入的内容放到数据总线上;然后发出“写”控制信号,在该信号的控制下,数据被写入指定的存储器单元中。

应当注意,写入操作将破坏该存储单元原存的内容,即由新内容代替了原存内容,原存内容将被清除。

【例 3-11】 以一个简单的加法运算为例,描述计算机的工作过程。

求解 5+8=? 的机器语言程序为

```
10110000  00000101        ;第 1 个操作数 (5)送到寄存器
00000100  00001000        ;5 与第 2 个数 (8)相加,结果 (13)送到寄存器
11110100                  ;停机
```

该段程序在内存中的存放形式如图 3-27 所示。由于读取每一条指令都是由一系列相同的操作组成,为简便起见,这里仅给出读取第一条指令的过程描述(如图 3-28 所示)。

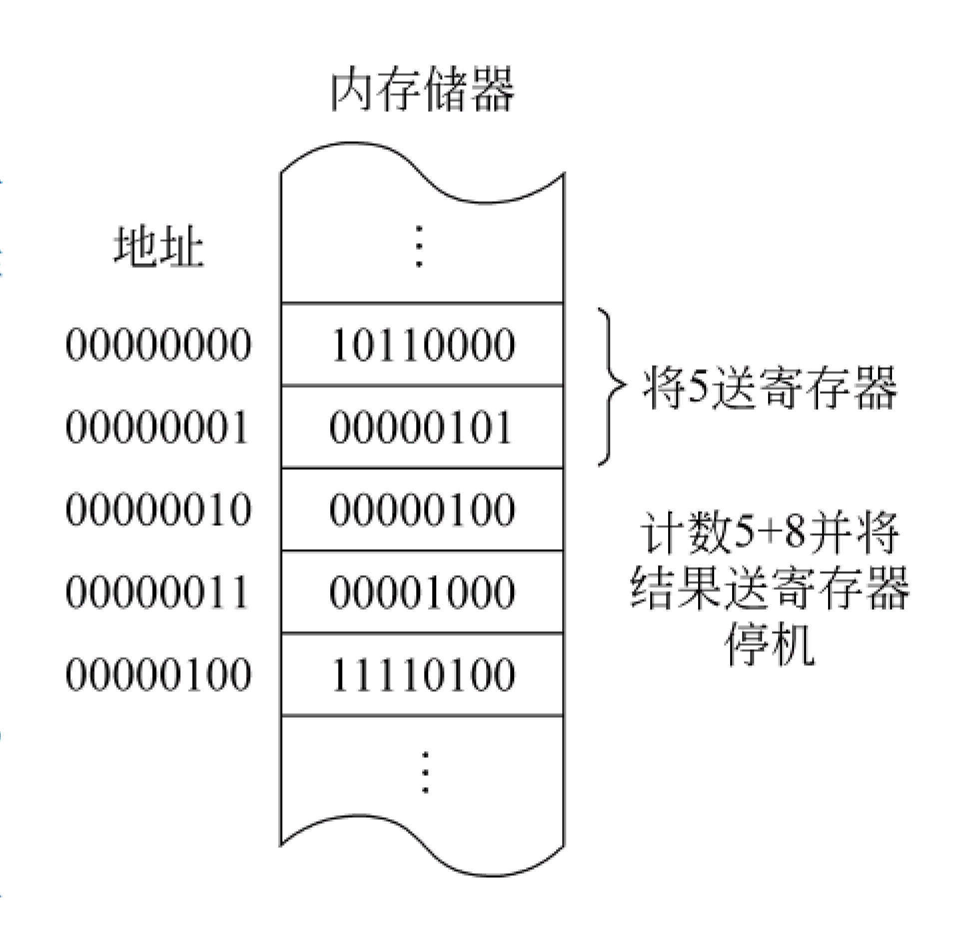


图 3-27 指令在内存中的存放形式

取第一条指令的过程如下:

(1) 将指令在内存中的地址(这里为 00000000)赋给程序计数器 PC,并送到地址寄存器 AR。

(2) PC 自动加 1(即由 00000000 变为 00000001),AR 的内容不变。

(3) 把地址寄存器 AR 的内容(00000000)放在地址总线上,并送至内存储器,经地址译码器译码,选中相应的 00000000 单元。

(4) 控制器发出读命令。

(5) 在读命令控制下,把所选中的 00000000 单元中的内容即第 1 条指令的操作码 10110000 读到数据总线。

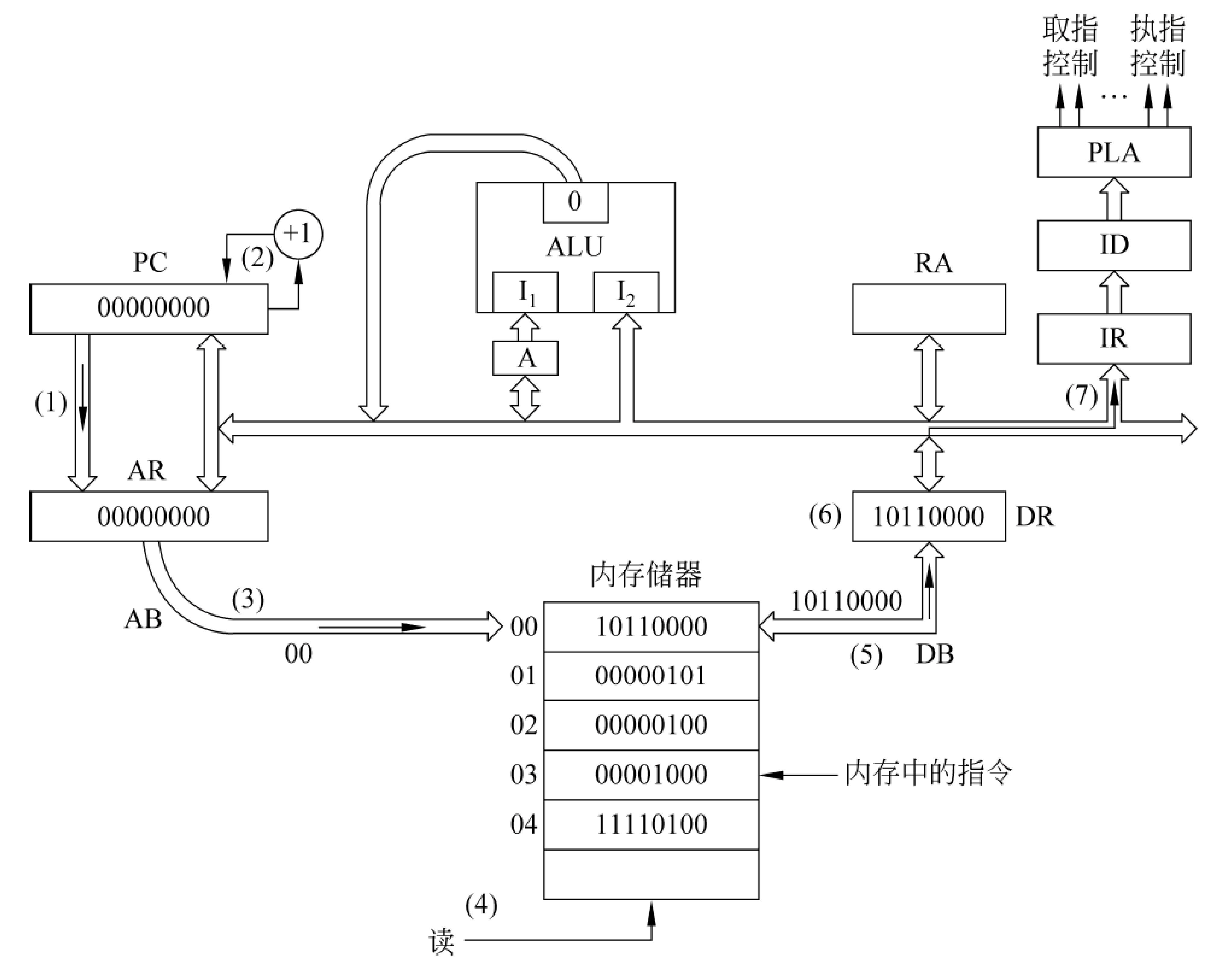


图 3-28　读取一条指令操作码的过程

(6) 把读出的内容 10110000 经数据总线送到数据寄存器 DR。

(7) 取指阶段的最后一步是指令译码。因为取出的是指令的操作码,故数据寄存器 DR 把它送到指令寄存器 IR,然后再送到指令译码器 ID。

读取存放在内存中的操作数的过程与取指令类似,仅第(7)步有不同。上例中,因指令要求读取的操作数要送到寄存器,故由数据寄存器 DR 取出的内容就通过内部数据总线送到寄存器中。由于运算的结果存放在处理器内部的寄存器,所以不需要再访问内存。

但需要注意的一点是,CPU 内部寄存器只能用于数据(中间运算结果)的暂时存放,最终的结果还是需要存放到存储器中。

3.4.3　图灵机与计算机

从以上的叙述,我们对现代计算机的基本工作原理已经有了初步的了解。现在来讨论第 1 章中介绍的图灵机与计算机的关系,以说明(仅仅是说明,而非严格的推理)为什么图灵模型是现代计算机的理论基础,或者说计算机只是图灵机的翻版。

先来看一个简单示例。

【例 3-12】 求 3 个正整数 a、b、c 之和。

这是一道小学数学题,求解该题目的方法可以说有无数种,例如:

$a+b+c$

$a+c+b$

$b+c+a$

$a+b+c+1-1$

……

很容易看出，不论用上述哪种方法，在 a、b、c 均为正整数的前提下，它们的计算结果都相同，或者说，它们是计算等价的。推而广之，对某一问题，如果用不同的求解方法都能得到相同的正确结果，那么就说这些方法是**计算等价**的。这就如同"条条大路通北京"一样，都从西安出发，走不同的路线，虽然花费的代价（时间、精力等）不同，但最终都到达了北京。这些路线就是计算等价的。如果能够说明图灵机与计算机计算等价，那么就可以说计算机只是图灵机的翻版。

回到例 3-12，已经显然地证明其中的各表达式相对于正整数是计算等价的，也就是说，既可以用其中一个来求解，也可以用另一个求解。换个角度，就是其中一个可以代替另一个，或者说，其中一个可以"模仿"另一个。这种"模仿"也称为"模拟"（这与第 2 章介绍的"模拟信号"可是两回事）。以上各表达式之间就可以相互模拟。所以，计算等价的前提是要能够相互模拟。即，如果集合 A 和 B 能够相互模拟，则 A 和 B 计算等价。以下就从什么是"模拟"入手，从不同角度说明图灵机与计算机的计算等价性。

1. 对应的功能部件

计算等价的前提就是要能够相互模拟。那么，什么是"模拟"呢？这是一个很难给出确切定义的名词。可以简单地说模拟就是一种模仿或是复制。比如有人对着你做了一个鬼脸，然后你也照着他的样子对他做了个鬼脸，这是你在模仿他，或者说你对他进行了模拟。考虑一下为什么你能够模拟他。重要的一点：因为他有手，你也有手，你的手对应他的手；他有眼睛，你也有眼睛，你的眼睛对应他的眼睛；你的手、眼睛和嘴的动作对应着他的手、眼睛和嘴的动作……即你们之间存在一系列的对应关系。

因此，**A 能够模拟 B 的关键条件是要具有对应关系**：如果 A 中元素可以完全对应 B 中元素，那么 A 就可以模拟 B（请注意：这句话隐含了在此条件下，B 不一定能模拟 A）。反之，如果 B 中元素也可以完全对应 A 中元素，B 也就可以模拟 A。

由 1.2.1 节已知，图灵机模型的 4 要素是无限长的纸带、读写头、输入输出控制规则及内部状态集合。如果将图灵机设为 A，计算机设为 B，那么，A 中的 4 大元素在 B 中都具有了一一对应的关系：纸带对应计算机中的存储器（不要想着计算机中的存储器容量是有限的，如果加上光盘等脱机外存，存储容量就可以认为无限大）；读写头对应计算机中的运算器及输入输出设备；控制规则显然就对应计算机中的程序；与图灵机一样，计算机中也有状态信息集合。由此可以说，计算机与图灵机具备了相互模拟的条件。

2. 信息变换（计算）能力

【例 3-13】 假设有 A 和 B 两个人，A 对着 B 做了个鬼脸，但 B 没有对着 A 做鬼脸，而是将 A 的鬼脸动作记在了日记本上。几天后，C 根据 B 在日记本上的描述记录，对着其他人做了鬼脸，与 A 的完全一样。

这里，C 将日记本上的文字翻译成了动作，完成了对 A 的模拟。这个"翻译"的过程

就是对信息的变换，而变换本身可以理解为一个计算。1.2.3 节中已给出了关于计算的详细描述。广义地讲，计算就是对信息的变换。例如，将 x 变换为 $f(x)$ 就是一种计算。这里的 x 是输入，$f(x)$ 则是输出。图灵机是一个计算装置，计算机也是一个计算装置，因为它们都可以将输入信息进行变换后给出相应的输出。

如果将一个图灵机对纸带信息的变换结果输入给另一台图灵机，然后再输入到别的图灵机……这样相当于对计算进行了组合，以实现更加复杂的计算（所以，千万不要以为图灵机只能像 1.2 节中描述的那样做简单计算）。如果一个简单计算对应一个图灵机，那么多个简单图灵机就可以构造出复杂的图灵机（如同 3.2 节中介绍的“封装”和“抽象”理论）。最简单的计算就是对 0 和 1 的运算。任何图灵机都可以把输入和输出信息用二进制进行编码，任何一种变换也可以最终分解为对 0、1 编码的变换，而对 0、1 编码的所有计算都可以分解为与、或、非 3 种基本逻辑运算，即用逻辑电路可以组合出任意的图灵机。

在式（1.1）的图灵机形式化描述中，有一个停机状态 F，它是内部状态 Q 的子集。当图灵机的工作碰到停机状态时就结束计算。

计算机也由各种逻辑电路组合而成，可以进行各种复杂的信息变换，而且在满足一定条件时可以停止。

3. 通用性

如果将例 3-13 中的 A 和 B 设为两台图灵机，按照能够相互模拟的条件，首先它们应具有一一对应的关系。因为都是图灵机，这一点是肯定的；除了对应关系，A 和 B 能否相互模拟还取决于它们是否计算等价。即对给定的输入，是否具有相同的变换（计算）结果。如果是，则就认为 A 机和 B 机可以相互模拟。

若设图灵机 A 的输出为 O，假如 B 的输出为 O'，为了使 B 能模拟 A，再通过一台图灵机 C，能够将 O' 变换为 O，那么就相当于 B 模拟 A 了（如图 3-29 所示）。

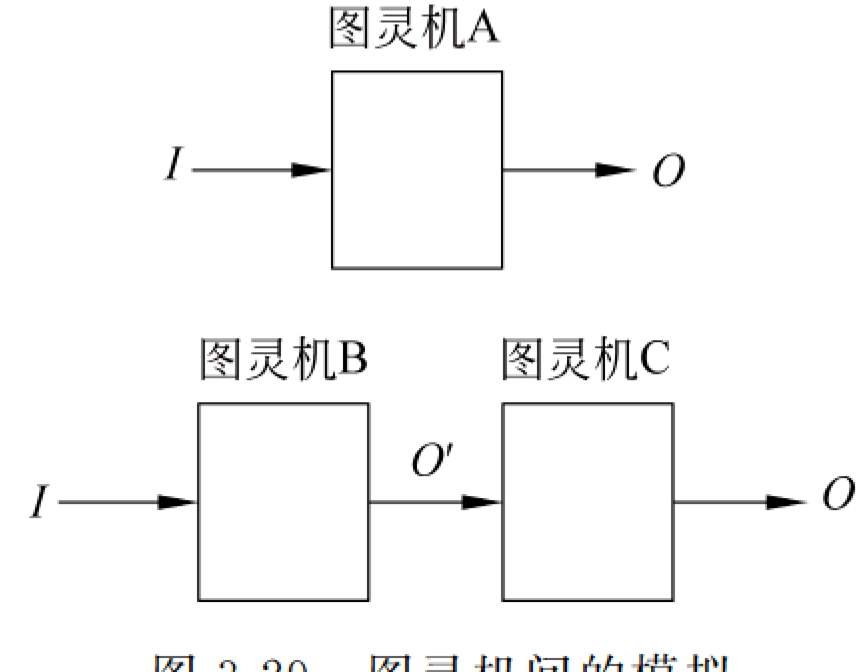


图 3-29　图灵机间的模拟

如果图灵机 A 能够模拟图灵机 B，并且 B 也能模拟 A，则说 A 和 B 是计算等价的。能够模拟其他所有图灵机的图灵机就称为**通用图灵机**（Universal Turing Machine，UTM），它能接受一段描述其他图灵机的程序，并运行程序实现该程序所描述的算法。这句话的简单解释是：任意一台图灵机 TM，其当前的“内部状态、输入数据、输出动作、下一时刻的内部状态”等信息都可以用 0 和 1 的组合（即编码）来表示，这样的一组编码就可以代表 TM。若该 TM 能够将输入 x 变换为输出 y，那么，将 TM 的编码及 x 输入到 UTM 中，则也会得到同样的 y。

以上从功能部件的对应性、信息变换能力及通用性 3 个方面叙述了图灵机与计算机的计算等价性，事实上，现代电子计算机就是这样一种通用图灵机的模拟。图 3-30 给出了一个多（纸）带图灵机模拟计算机的示意图（多带图灵机可以采用固定的模式转换为单带的图灵机，具体的转换方法可查阅其他详细介绍图灵机的参考书）。

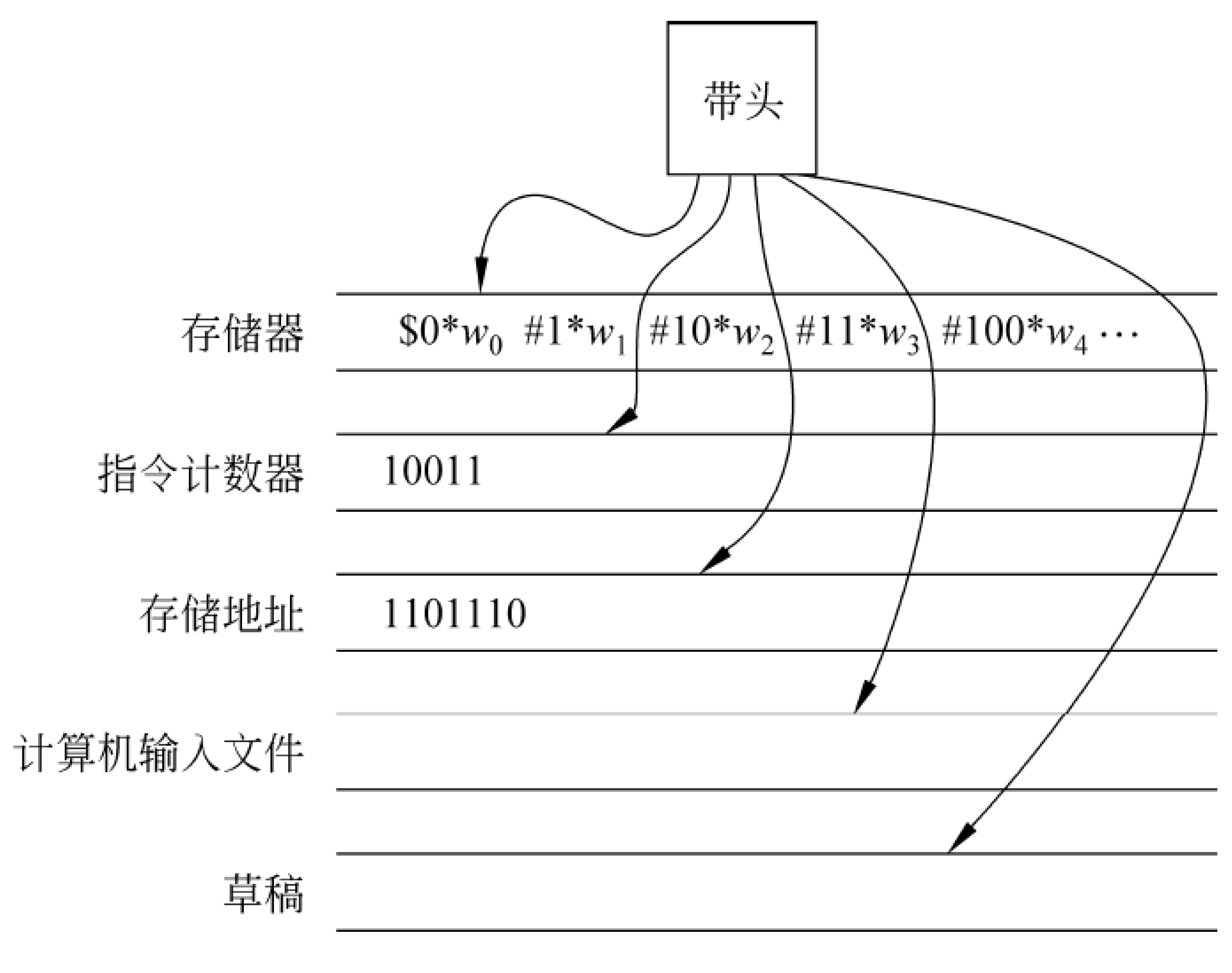


图 3-30　图灵机模拟计算机

第一条带(带 1)表示计算机的存储器。假设存储单元的地址按照数值顺序与这些存储单元中的内容交替出现。地址和内容都用二进制书写。符号 * 和 # 用来表示地址和内容结尾,以及区分二进制串是地址还是内容。另一个标记 $ 表示地址和内容序列的开头。如 0 * 表示这里的 0 是地址,而 w_0 # 表示 w_0 是存放的内容。

第二条带(带 2)是“指令计数器”(相当于程序计数器 PC)。这条带保存一个二进制整数,它表示第一条带上的一个存储单元的地址,而该存储单元中的内容是将要执行的下一条计算机指令。

第三条带(带 3)保存数据的“存储地址”或这个地址的内容(当在带 1 上确定地址位置之后)。为了执行指令,图灵机必须找到一个或多个保存着计算所涉及数据的存储地址的内容。首先,把所需地址复制到带 3 上并与带 1 上的地址比较,直到发现匹配为止。将带 1 上地址中对应的内容复制到带 3 上,并移动到所需要的任何地方,典型情况是,移动到表示计算机寄存器的一个低编号地址。TM 将模拟计算机的指令周期如下:

(1) 搜索带 1,寻找与带 2 上指令号匹配的地址。由于带 2 上保存的是下一条要执行的指令的地址,从带 1 上 $ 处开始,向右移动,比较每个地址与带 2 的内容。比较的过程是使带头一前一后地向右移动,并验证扫描的符号是否相同。

(2) 找到指令地址时检查地址中的内容(指令译码)。由于内容是指令,则其前几个位(指令码)表示要做的动作(比如复制、加、分支等),剩余位表示动作中涉及的一个或多个地址(操作数)。

(3) 如果指令的操作数是地址码(即运算对象的存放地址),则这个地址会被复制到带 3 上。同时将指令的位置标记到带 1 的第二道(在图 3-30 中没有显示出来),以在必要时能回到这条指令。

(4) 执行指令。

(5) 在执行指令并确定指令不是跳转之后,给带 2 上的指令计数器加 1,再次开始指令循环。

图灵机如何模拟典型计算机,还有许多其他细节。在图 3-30 中显示了第四条带,这

条带保存被模拟的计算机输入，因为计算机必须从文件读输入，图灵机可以改为从这条带来读取输入。图 3-30 还显示了一条草稿带。模拟有些计算机指令可能有效地使用一条或多条草稿带来计算诸如乘法的算术运算。

以下是图灵机实现数据传送操作的一个示例描述，读者可以对比微型机的工作过程。把一个数(源操作数)传送到某个地址(目标地址)中的操作过程如下：

(1) 从指令中得到这个源操作数的地址。

(2) 把这个地址写在带 3 上。

(3) 在带 1 上搜索这个地址，即找到要传送的数(源操作数)。

(4) 用同样的方法找到目标地址后，把这个源操作数复制到为目标地址保留的空间里，就实现了"传送"。

如果需要更多的空间来保存这个源操作数，或者源操作数比目标地址中原来的值占用更少的空间，则可以通过平移来改变可用的空间。即：

(1) 把新的值所占之处右边的整个非空白带复制到草稿带上。

(2) 把新的值写下来，使用这个值的正确的空间数量。

(3) 把草稿带重新复制到带 1 上，紧接着新值的右边。

还可能出现的一种特殊情形是：这个目标地址可能还没有出现在带 1 上，因为在此之前计算机可能还没有用到过它。此时，就在带 1 上找到源数据所属的地方，平移腾出适当的地方，把地址和新的值都保存在这个地方。

最后，假设计算机能够"接受"输出确认指令(可能对应着计算机调用的往输出文件上写 yes 的函数)。当图灵机模拟这条计算机指令的执行时，图灵机进入自身的接受状态并停机。

上面的讨论远远不是完整的形式化的图灵机与计算机相互模拟的证明，但它应当提供了足够的细节来说明图灵机是计算机能够做什么的有效表示。可以只使用图灵机作为任意种类的计算机的模拟算装置，通过该装置，更好地研究计算机能计算什么并给出严格的表示。

最后给出断言：计算机能够模拟图灵机，图灵机也能够模拟计算机，即计算机与图灵机是计算等价的，或者说计算机只是图灵机的翻版。所以说图灵机是计算机的理论模型。

3.4.4 冯·诺依曼结构的局限性

冯·诺依曼的"存储程序计算机结构"为计算机技术的发展做出了巨大的贡献，几十年来，虽然计算机技术有了迅猛的发展，但传统计算机依然采用的是冯·诺依曼的体系结构。

传统的冯·诺依曼计算机结构属于控制驱动方式。它由存放在内存中的程序指明计算机的操作内容，指令的执行顺序受程序计数器的控制(如 3.4.2 节所述)，也就是说由指令控制器控制指令执行的顺序和时机，当它指向某条指令时才驱动该条指令的执行。这种结构的特点是"程序存储，共享数据，顺序执行"。计算中有一条单一的控制流从一条指令传到下一条指令(由程序计数器 PC 提供，执行 K、$K+1$、…、N 指令)，执行指令所需要的操作数通过指令中给定的地址来访问，指令执行结果也通过地址存入一个共享的存储

器中。因此，存储程序工作方式的冯·诺依曼计算机本质上是顺序处理机，它的软件和硬件完全分离，适合对确定的算法和数据进行数值计算，对非数值的处理就显得不足。

随着计算机应用领域的不断拓展，对计算机的性能特别是对非数值数据的处理能力提出了更高的要求。如各类3D模型的计算和处理、气象信息处理等，都要求计算机的计算能力能达到万亿次每秒以上。这就使得传统的冯·诺依曼计算机难以满足这种需求，逐渐暴露出其体系结构上存在的不足，主要表现在以下几方面：

（1）由于CPU与存储器之间会有大量的数据交互，而总线的传输能力却有限。因此使系统的性能受到了总线传输能力的制约，造成总线瓶颈。

（2）按照存储程序原理，指令的执行顺序由程序决定。这就要求在编写程序时必须仔细地分析任务的处理顺序。这对一些大型的、复杂的任务是比较困难的（即需要准确地做好需求分析和模块设计）。

（3）由于指令的执行顺序由程序计数器控制，使得即使有关数据已经准备好，也必须逐条执行指令序列。提高计算机性能的根本方向之一是并行处理，而冯·诺依曼计算机难以实现真正的并行处理。

（4）以运算器为中心，I/O设备与存储器间的数据传送都要经过运算器，使处理效率特别是对非数值数据的处理效率比较低。

（5）冯·诺依曼计算机具有简单的逻辑运算和判断功能，但远不能适应复杂的问题求解和推理的要求。

由于在体系结构上存在以上这些局限，从根本上限制了计算机特别是并行计算的发展。因此，从20世纪80年代起，陆续提出了多种与冯·诺依曼计算机截然不同的新概念模型的系统结构。如并行计算机、数据流计算机、量子计算机、生物计算机等非冯·诺依曼计算机，它们部分或完全不同于传统的冯·诺依曼计算机，在很大程度上提高了计算机的计算性能。

*3.4.5 哈佛结构

由于计算机采用二进制，指令和数据都是用二进制码表示，指令和操作数的地址又紧密相关，因此，冯·诺依曼结构很自然地就采用了将指令和数据统一存放，并共享同一组总线。但这种结构使得信息流的传输只能采用串行方式，从而影响了计算机性能的提高。例如，我们已经知道，完成一条指令的执行需要经过取指令、指令译码、读取操作数、执行和存放结果这样5个步骤。由于指令和数据存放在同一存储器中，共用同一条总线（数据总线）传输，使取指令时就必然无法同时读取操作数。因此，它们无法重叠执行，只能串行执行。

哈佛结构（Harvard architecture）是一种并行体系结构，其结构模型如图3-31所示。它将程序指令和数据分开存储在不同的存储空间中，即程序存储器和数据存储器是两个独立的存储器，每个存储器独立编址、独立访问。相应地就有4条

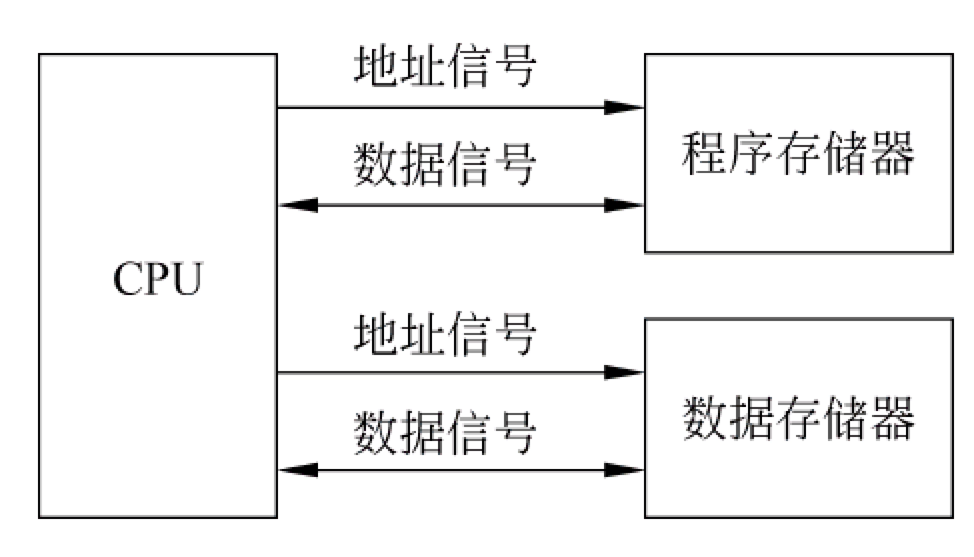


图3-31 哈佛结构模型

系统总线：用于传送指令的数据总线和地址总线，及用于传送数据的数据总线和地址总线。这种分离的程序总线和数据总线可以允许在一个机器周期内同时获得指令操作码（来自程序存储器）和操作数（来自数据存储器），使数据的吞吐率提高了1倍，从而提高了计算机的执行速度。另外，由于程序和数据存储在两个分开的物理空间中，因此取指令和存取操作数可以重叠执行。处理器首先到指令存储器中读取指令，经过译码后得到数据地址，再到相应的数据存储器中读取数据，并进行下一步执行指令的操作。

总之，与冯·诺依曼结构相比，哈佛结构具有如下两个显著特点：

(1) 使用两个独立的存储器模块，分别存放指令和数据。每个模块中都不允许指令和数据并存。

(2) 使用独立的两组总线，分别作为CPU与存储器之间的专用指令和数据的通信通道。这两组总线间毫无关联。

在改进的哈佛结构中，将图3-31中的两组总线合并为一组，公共地址总线用于访问两个存储器模块，公共数据总线则被用来完成程序存储器或数据存储器与CPU之间的数据传输，两条总线由程序存储器和数据存储器分时共用。

在现代处理器中，程序存储器和数据存储器均采用cache，省去了从主存储器中读取指令和数据的时间，大大提高了运行速度。

冯·诺依曼结构和哈佛结构的主要区别就在于程序空间和数据空间是否为一体。前者将两个空间重合，而后者是分开的。

冯·诺依曼计算机简单、低成本的总线结构，造就了计算机特别是微型计算机的迅速发展。目前，虽然已有众多关于非冯·诺依曼计算机的研究，但数据流计算机主要用于大型机系统中，其他如量子计算机、生物计算机等都还处于实验室研究阶段，离进入市场还有相当的距离。

在现代微型计算机系统中，其总体结构依然是冯·诺依曼结构，但在微处理器内部，由于采用cache技术，实现了指令和数据分开存放，同时共享公共总线，属于改进型的哈佛结构。与冯·诺依曼结构相比，哈佛结构复杂度比较高，对外围设备的连接和处理要求较高，不适合存储器扩展。所以，除了在CPU内部之外，哈佛结构主要应用于单片机和微控制器中，如Intel公司的51系列、Microchip公司的PIC16、ARM公司的ARM9～ARM11等。

3.5 操作系统

操作系统(Operating System，OS)是管理计算机硬件与软件资源的程序，同时也是计算机系统的内核与基石。没有它，今天的计算机可以说是完全没有意义的。

3.5.1 操作系统概述

操作系统在计算机系统中的作用相当于“大脑”在人体中的作用。不论这种比喻是否

恰当，但至少说明了一个问题——操作系统对计算机系统而言是至关重要的。

1. 什么是操作系统

首先，操作系统是一个程序，虽然它非常庞大和复杂，但它依然只是一个程序，是控制其他程序运行、管理系统资源并为用户提供操作界面的系统软件。

操作系统由一系列具有不同管理和控制功能的程序模块组成，位于硬件和用户之间，是覆盖于计算机硬件系统上的第一层软件。它一方面为用户提供接口，方便用户使用计算机；另一方面它能管理计算机软硬件资源，以便合理充分地利用它们。

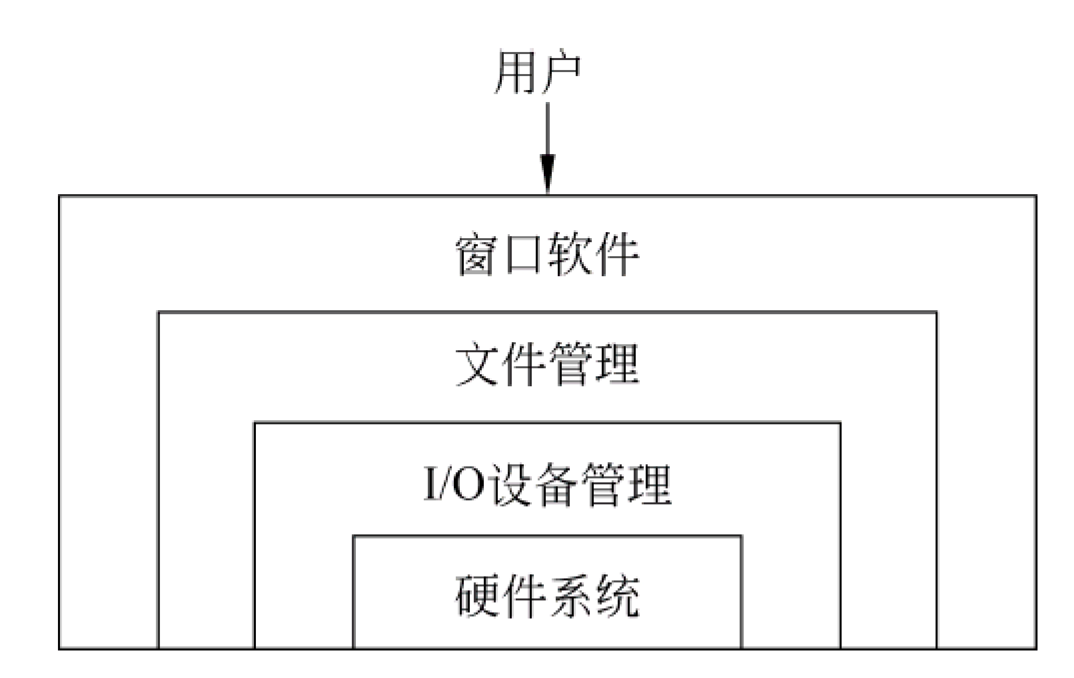


图 3-32　操作系统的作用示意图

操作系统的作用可以用图 3-32 示意，总体上包括以下几个方面：

（1）隐藏硬件。由于直接对计算机硬件进行操作非常困难和复杂，因此，从用户的角度，需要计算机具有友好、易操作的使用平台。

（2）为用户和计算机之间的“交流”提供统一的界面，使用户不必考虑不同硬件系统可能存在的差异。

（3）管理系统资源。计算机系统中的主要资源有处理器、存储器、I/O 设备、运行的数据和程序。资源管理主要就是对以上 4 种资源有效地进行的管理和分配，使有限的系统资源能够发挥更大的作用。

早期的计算机中没有操作系统，用户在计算机上的操作完全由手工进行，采用绝对的机器语言（二进制代码）形式编写程序，通过接插板或开关板控制计算机操作。这个时期的计算机只能一个个、一道道地串行算题，一个用户上机，就独占了全机资源，使资源利用率和效率都很低。

晶体管的诞生使得计算机产生了一次革命性的变革。操作系统的初级阶段是监控程序和批处理程序。在 20 世纪 60 年代早期，商用计算机制造商制造了批处理系统，此时不同型号的计算机具有不同的操作系统，无通用性。

1964 年，第一代共享型、而非每种产品量身定做的操作系统 OS/360 诞生，它可以运行在当时 IBM 公司的系列大型计算机上。

随着计算机技术的发展，操作系统的功能越来越强大。今天的操作系统已有包括分时、实时、并行、网络、及嵌入式操作系统等多种类型，成为不论大型机、小型机还是微型机都必须安装的系统软件。

2. 操作系统的分类

对操作系统进行严格的分类是困难的。早期的操作系统，按用户使用的操作环境和功能特征的不同，可分为 3 种基本类型：批处理系统、分时系统和实时系统。随着计算机体系结构的发展，又出现了嵌入式操作系统、分布式操作系统、个人计算机操作系统和网络操作系统。

目前的操作系统种类繁多,很难用单一标准统一分类。若从应用领域划分,可分为桌面操作系统、服务器操作系统、主机操作系统和嵌入式操作系统等;若根据所支持的用户数目,可分为单用户系统(如 Windows、MS-DOS 等)和多用户系统(如 UNIX 等);从硬件结构的角度,可分为网络操作系统(如 NetWare、Windows NT 等)、分布式操作系统(如 Amoeba 等)和多媒体操作系统(如 Amiga 等)。除此之外,还可以从源码开放程度、使用环境、技术复杂程度等多种不同角度进行分类。下面简要介绍几种类型的操作系统。

1) 分时操作系统

分时操作系统(time-sharing operating system)是指多用户通过终端共享一台主机 CPU 的工作方式。为使一个 CPU 为多道程序服务,将 CPU 划分为很小的时间片,采用循环轮作方式将这些 CPU 时间片分配给排队队列中等待处理的每个程序(如图 3-33 所示)。由于时间片划分得很短,循环执行得很快,使得每个程序都能得到 CPU 的响应,好像在独享 CPU。分时操作系统的主要特点是允许多个用户同时运行多个程序,每个程序都是独立操作、独立运行、互不干涉。现代通用操作系统中都采用了分时处理技术。例如,UNIX 是一个典型的分时操作系统。

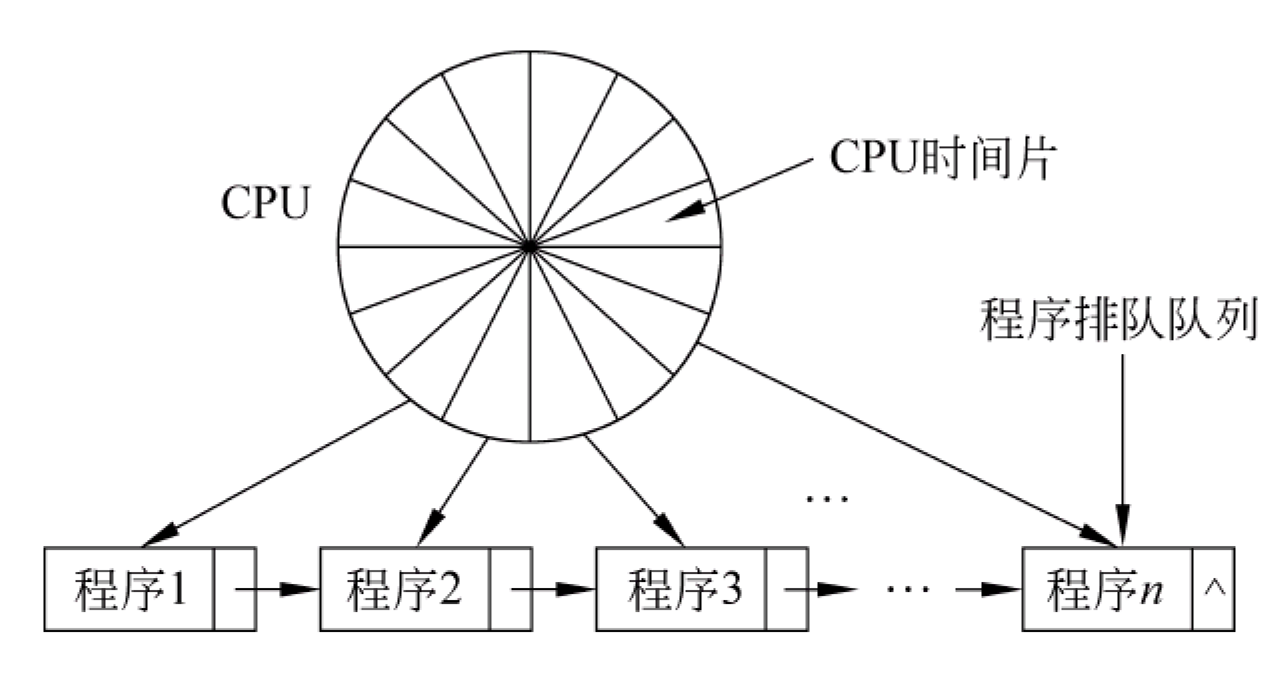


图 3-33　分时占用 CPU 时间片示意图

2) 网络操作系统

网络操作系统(Net Operating System,NOS)是向网络计算机提供服务的特殊的操作系统。它在计算机操作系统下工作,使计算机操作系统增加了网络操作所需要的能力。NOS 运行在网络中称为服务器的计算机上,并由联网的计算机用户(客户端)共享,它的功能包括网络管理、通信、安全、资源共享和各种网络应用。网络操作系统的目标是用户可以突破地理条件的限制,方便地使用远程计算机资源,实现网络环境下计算机之间的通信和资源共享。例如,Novell NetWare 和 Windows NT 就是网络操作系统。

3) 分布式操作系统

分布式操作系统(distributed software system)是指通过网络将大量计算机连接在一起,以获取极高的运算能力、广泛的数据共享以及实现分散资源管理等功能为目的的一种操作系统。它的优点是:①分布性。它集各分散结点计算机资源为一体,以较低的成本获取较高的运算性能。②可靠性。由于在整个系统中有多个 CPU 系统,因此当某一个 CPU 系统发生故障时,整个系统仍旧能够工作。显然,在对可靠性有特殊要求的应用场合可选用分布式操作系统。

4）个人计算机操作系统

个人计算机操作系统是一种单用户的操作系统。它的特点是计算机在某一时间为单个用户服务。现代个人计算机操作系统采用图形界面人机交互方式操作，用户界面友好，用户无须学习专业理论知识，就可以掌握对计算机的操纵。典型的个人计算机操作系统是 Windows。

3. 操作系统的功能

操作系统的职能是负责系统中软硬件资源的管理，合理地组织计算机的工作流程，并为用户提供一个良好的工作环境和友好的使用界面。

从资源管理角度看，操作系统具有五大基本功能（如图 3-34 所示）：

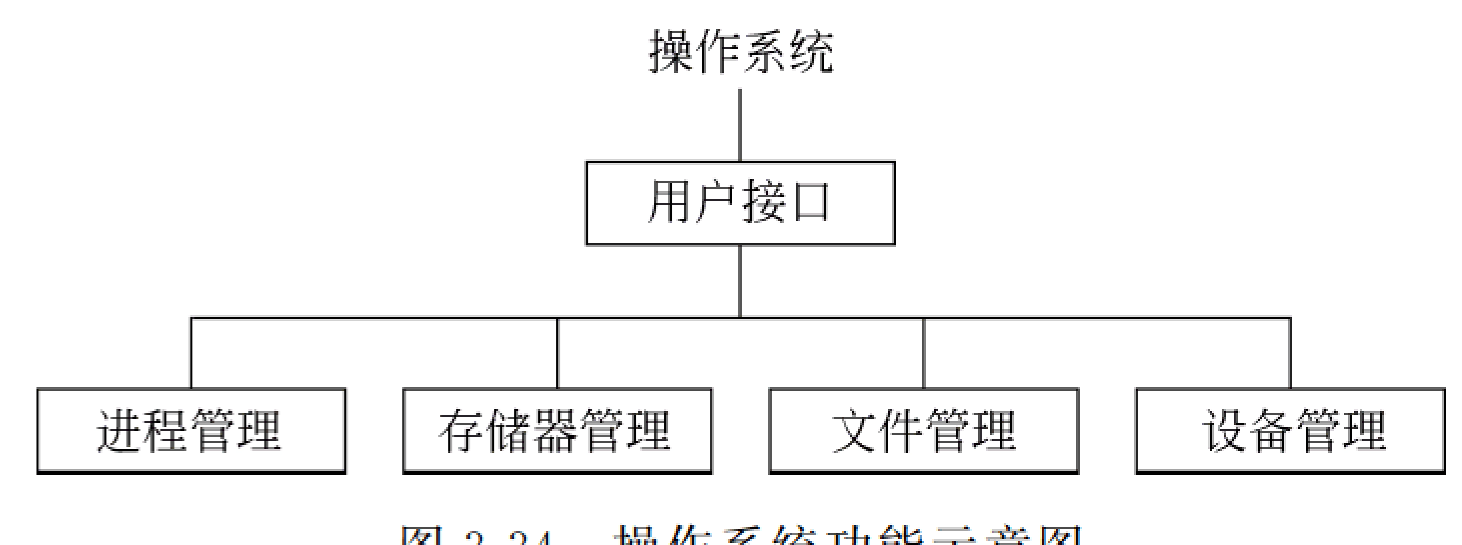


图 3-34　操作系统功能示意图

（1）**进程管理**。又称作业管理或处理器管理，其主要任务是对处理器的时间进行合理分配，对处理器的运行实施有效的管理。

（2）**存储器管理**。由于多道程序共享内存资源，所以存储器管理的主要任务是对存储器进行分配、保护和扩充。

（3）**文件管理**。有效地管理文件的存储空间，合理地组织和管理文件系统，为文件访问和文件保护提供更有效的方法及手段。

（4）**设备管理**。根据确定的设备分配原则对设备进行分配，使设备与主机能够并行工作，为用户提供良好的设备使用界面。

（5）**用户接口**。用户操作计算机的界面称为用户接口（或用户界面），用户通过命令接口或程序接口（Application Programming Interface，API），实现各种复杂的应用处理。

3.5.2　处理器管理

处理器管理又称进程管理或作业管理，其主要任务是对处理器进行分配（解决谁来使用处理器和怎样使用处理器的问题），并对其运行进行有效的控制和管理。在操作系统中，把用户请求处理器完成一项完整的工作任务称为一个作业。当有多个用户同时要求使用处理器时，允许哪些作业进入，不允许哪些进入，怎样安排已进入作业的执行顺序等，这些就是处理器管理模块的任务。由于在多道程序环境下①，处理器的分配和运行都是

① 多道程序环境是指在计算机中有多个程序在同时运行，也称并发执行。现代计算机都采用多道程序并发执行方式。

以进程为基本单位的，所以对处理器的管理最终可以归结为对进程的管理，包括进程控制、进程同步、进程通信和进程调度。本节仅涉及进程管理的一般描述，详细的控制过程和算法请参阅操作系统相关书籍。

要理解进程管理，首先要了解“进程”这个概念，而要弄清楚什么是进程，又需要从程序的执行方式讲起。

1. 程序的执行方式

由 3.3.1 节已知，程序是实现某一特定功能的指令序列。这里的“功能”可能是由若干个“小功能”(程序段)组成的。程序在执行时，必须按照一定的次序，只有在前一个程序段执行完，后一个程序段才能进行。比如，只有输入了用户程序和数据，才能进行计算(处理)，然后才能输出。没有第 1 步，也就无法进行第 2 步。以下为了叙述方便，假设一个程序中包括输入(I)、计算(C)和打印输出(P)3 个程序段，用结点(node)表示程序段的操作，则程序的执行过程如图 3-35 所示。

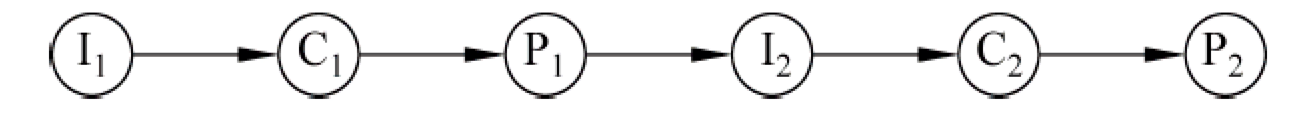


图 3-35　程序顺序执行方式示意图

图 3-35 中的输入、计算和打印输出 3 个程序段之间存在着严格的执行顺序，只有输入了信息才能进行计算，只有计算产生了结果才能输出。即对一个作业，存在着 $I_1 \rightarrow C_1 \rightarrow P_1$ 这样的前趋关系，必须按顺序执行。

【例 3-14】 假设有 3 条语句的程序如下：

S_1: $a=x+y$
S_2: $b=a+4$
S_3: $c=b+1$

该 3 条语句必须按 $S_1 \rightarrow S_2 \rightarrow S_3$ 的顺序执行。因为只有在 a 被赋值后才能执行 S_2，b 被赋值后才能执行 S_3。

图 3-35 所示的方式称为程序的顺序执行方式。顺序执行时，任一时刻系统中只有一个程序在执行，程序工作于封闭环境中，独占系统的全部资源，只要环境和初始条件相同，程序无论执行多少遍，结果都是一样的。所以，顺序执行具有顺序性、封闭性、结果可再现性等特点。这种特性很便于程序员对程序进行检测和校正，但降低了资源的利用率和系统的处理效率。

虽然顺序执行有它独有的优点，但难以适应现代多任务系统的要求。再来看图 3-35。对一个作业，执行过程存在 $I_1 \rightarrow C_1 \rightarrow P_1$ 这样的顺序，但并不说明存在 $P_i \rightarrow I_{i+1}$ 的关系，也就说并不是第 2 个作业的输入必须取决于第 1 个作业的输出(这个前提很重要)。因此，在对多个作业进行处理时，可以不按照图 3-35 所示的顺序执行方式，而是在第 1 个程序段输入后开始进入计算阶段时，就可以输入第 2 个程序段，第 1 个程序段计算完开始输出时，就可以同时开始计算第 2 个程序段，并输入第 3 个程序段……这样多道程序同时执行的方式称为并发执行方式，可用图 3-36 表示。

图 3-36 说明，I_{i+1} 和 C_i 及 P_{i-1} 是重叠的，即它们是可以同时执行的。对这种并发执行方式也来看一个示例。

【例 3-15】 执行以下程序语句：

S_1：a=x+2

S_2：b=y+4

S_3：c=z+9

S_4：d=a+b+c

可以看出，S_1、S_2 和 S_3 可以同时（并发）执行，因为它们彼此互不依赖。只有 S_4 必须在 S_1、S_2 和 S_3 执行结束后才能执行（必须要等 a、b、c 被赋值）。该程序段的执行也可以用图 3-37 表示。

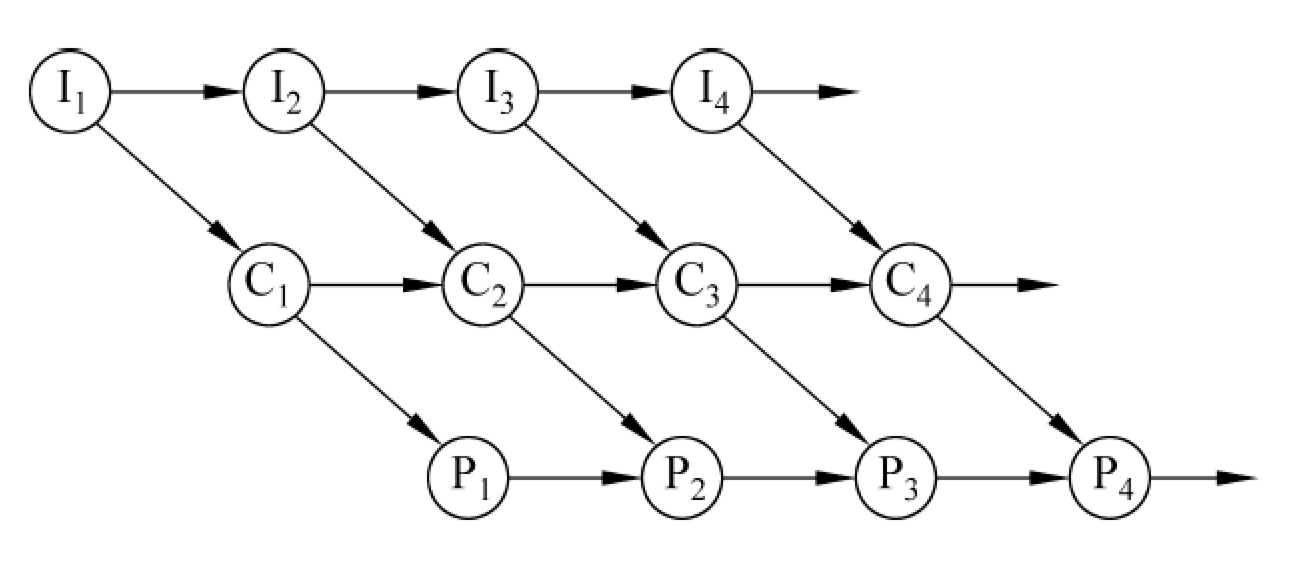


图 3-36　程序并发执行方式示意图

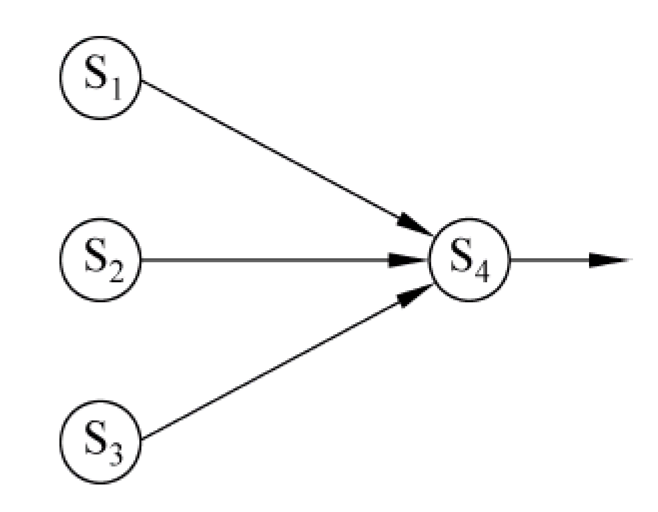


图 3-37　例 3-15 程序执行示意图

程序的并发执行，提高了系统的效率，但也产生了一些与顺序执行不同的新特征。首先，由于多个程序并发执行，整个系统资源为多程序共享，这样虽然提高了资源的利用率，但存在多个程序对资源的竞争和相互制约问题。例如，打印机被 P_1 占用，P_2 就不能再使用，只好等待，因此就存在“走走停停”的情况。由于多道程序共享系统中的各类资源，因而这些资源的状态将由多个程序来改变，程序的运行环境不再具有“封闭”性，亦即程序的执行会受到其他程序的影响，从而也导致失去了可再现性，造成多个程序可能因执行的先后顺序不同而得到不同的结果。

【例 3-16】 有两个循环执行的程序 A 和 B，共享一个变量 N。程序 A 每执行一次，都要做 N＝N＋1 的操作，程序 B 每执行一次都要做“输出 N，N＝0”操作。设某时刻 N＝n，且 A 和 B 以不同的速度运行，则可能会出现以下 3 种情况：

(1) 先运行 A，再运行 B，得 N 值为

n＋1，n＋1，0

(2) 先运行 B，再运行 A，得 N 值为

n，0，1

(3) 先运行 B 的“输出 N”，再运行 A，之后运行 B 的 N＝0，得 N 值为

n，n＋1，0

例 3-16 的执行结果说明，程序在并发执行时，由于失去了封闭性，其计算结果就不再如顺序执行方式那样可再现，而变得与它的执行速度有关。即，程序在经过多次执行后，虽然每次执行的环境和初始条件都一样，但会得到不同的结果。

2. 进程

为了使程序在多道程序环境下能并发执行,并确保可再现性,1966 年美国麻省理工学院的 J. H. Sallexer 提出了“进程”的概念。引入这一概念的目的就是为了解决现代计算机中多道程序共享系统资源的问题。

进程(process)被定义为“可并发执行的程序在一个数据集合上的运行过程”(这句话看上去有点高深)。简单地说,进程就是执行起来的程序。程序本身是静态的(编写好了可以存放在磁盘上不动),但进程是动态的,是“活”着的程序。有“活”当然就有“亡”,所以,进程是有生命周期的。进程的主要特征有以下几个:

(1) **动态性**。这是进程最基本的特征。表现在:因创建而产生,由调度而执行,因得不到所需资源而暂停,因被撤销而消亡。

(2) **并发性**。这是进程的重要特征,也是操作系统的重要特征。并发性是指多个进程同存于内存中,能在同一段时间内同时运行。

(3) **独立性**。进程是一个能独立运行的基本单位,也是进行资源分配和调度的独立单位。

(4) **异步性**。每个进程都按独立的、不可预知的速度向前推进,即,各进程并不是按相同的速度运行,而是异步运行,这一特征就导致了程序执行的不可再现性(如例 3-16)。为了避免这一点,操作系统中专门设置了一个称为“进程控制块”(Process Control Block,PCB)的数据结构,它负责记录进程的所有相关信息,保证进程从暂停到重新运行时能获得其暂停时的状态。

打开 Windows 中的任务管理器,就可以看到多个进程的运行情况。在图 3-38 中,左侧为启动的 Word、Excel 和计算器 3 个应用程序,右侧是系统运行的进程列表,有系统程序进程,也有应用程序进程,其中椭圆线中的是左侧 3 个应用程序对应的进程。

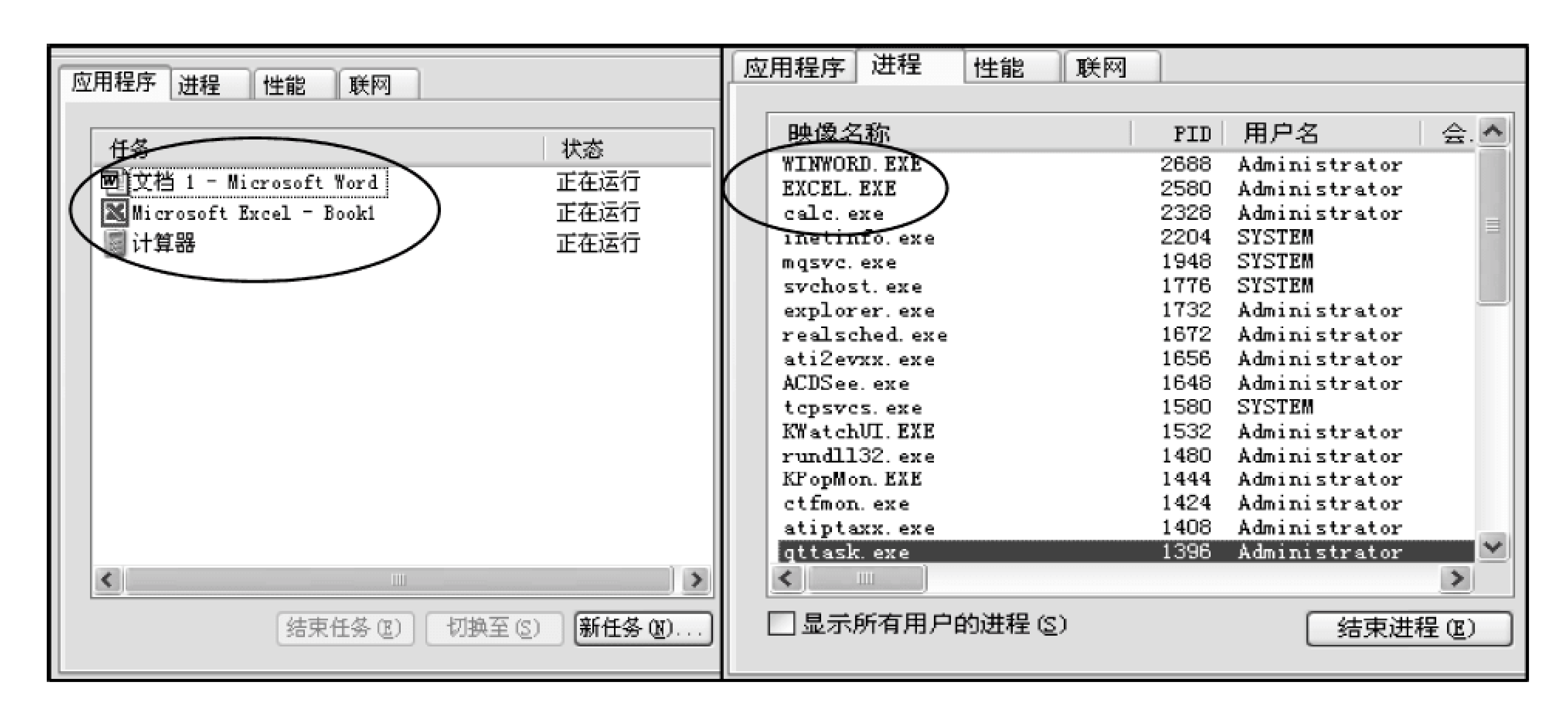


图 3-38　Windows 任务管理器中的应用程序及相应的进程列表

3. 进程的基本状态

上文已谈到,进程是“活着的程序”,因此它具有生存周期。在它“活着”的时间里,并不是始终处于运行状态(因为系统中不是只有一个进程),它会受到资源(如 I/O 请求等)

的制约。如果它需要的资源正被其他进程占用,它就只能停下来等待。所以,进程的执行过程是间断性的,其状态处于不断变化中。通常,一个进程必须具有以下 3 种基本状态(如图 3-39 所示):

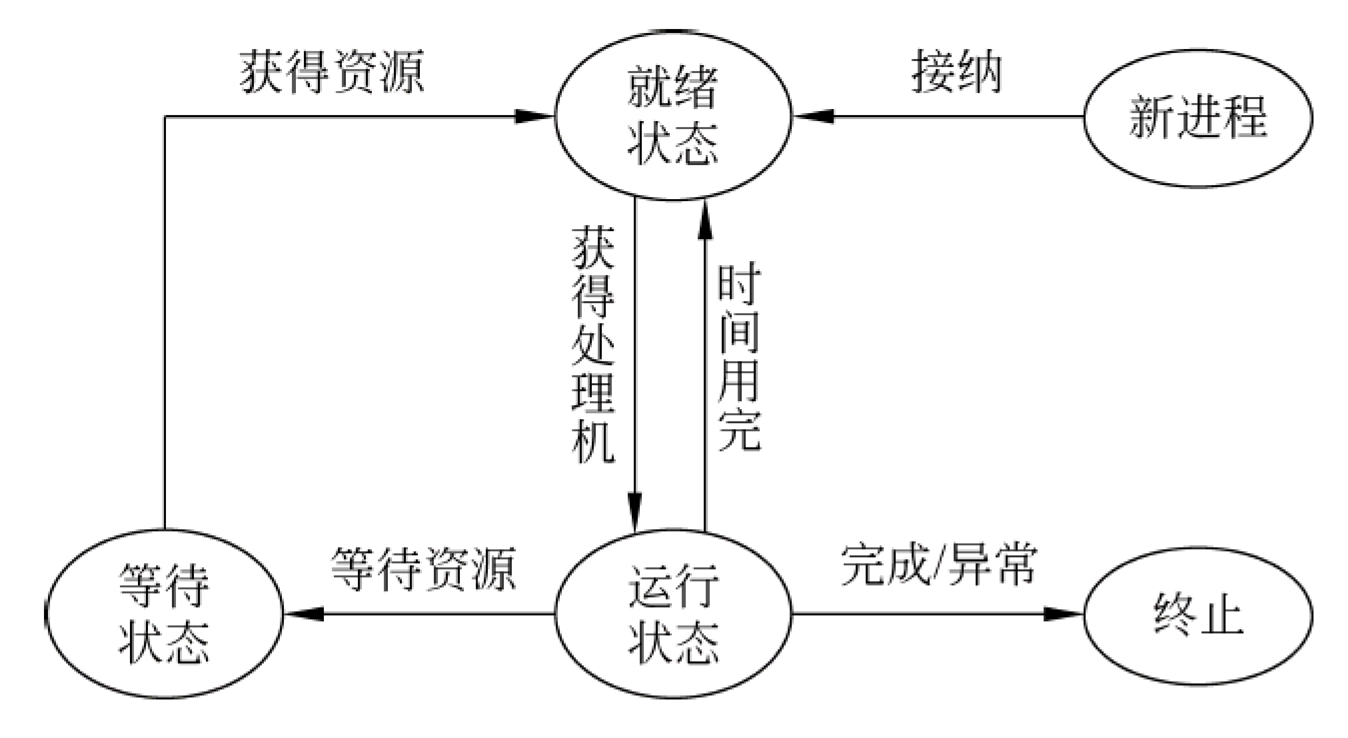


图 3-39 进程的 3 种基本状态

(1) **就绪(Ready)状态**。进程已经获得了除 CPU 之外所必需的一切资源,一旦分配到 CPU,就可以立即执行("万事俱备,只欠东风")。在多道程序环境下,可能有多个处于就绪状态的进程,通常将它们排成一队,称为就绪队列。

(2) **运行状态**。进程获得了 CPU 及其他一切所需资源,正在运行。对单个 CPU 系统而言,只能有一个进程处于运行状态;在多处理机系统中,则可能有多个进程处于运行状态。

(3) **等待状态**。由于某种资源得不到满足,进程运行受阻,处于暂停状态,等待分配到所需资源后,再投入运行。处于等待状态的进程也可能有多个,也将它们组成排队队列。

4. 进程控制和调度

进程控制的主要任务是调度和管理进程从"创建"到"消亡"整个生存周期过程中的所有活动,包括创建进程、转变进程的状态、执行进程、撤销进程等操作。

我们已经知道,进程具有就绪、运行、等待 3 种基本状态,在多数操作系统中还增加了"新状态"和"终止状态"两种。进程在它的整个生命周期中,就是不断地从一个状态转换到另一个状态,直到运行完成或出现异常结束,才进入终止状态。进程状态的转换过程可以简述如下:

- 操作系统创建一个新进程,先为其分配相应的资源,并使其处于"新状态"。
- 当就绪队列能够接纳新进程时(比如有空位了),操作系统就将其送入就绪队列,处于就绪状态。
- 操作系统根据进程在就绪队列中的位置、优先级等信息(这些都在 PCB 中)及进程调度算法,为进程分配 CPU,从而使其转入运行状态(此时的进程称为当前进程)。
- 运行中的进程可能因所需资源不能得到(或许该资源正在被其他进程使用)而无法进行执行,则转入等待状态(此时因为 PCB 中记录有各种"现场信息",再加上适当的进程同步控制手段,使例 3-16 中那样的"不可再现"情况不会出现了)。

• 运行中的进程也有可能虽然拥有所需的全部资源，但分给它的 CPU 时间用完了，或有更高优先级的进程出现，则只好退出 CPU，转入就绪队列。

• 执行完成或出现异常，则进程转入终止状态。

一个进程可以在 3 种基本状态之间多次转换，但新状态及终止状态只能出现一次。图 3-39 给出的是基于时间片轮转法的进程状态转换过程。

"进程调度"就是根据具体情况为每个进程分配 CPU。不同的操作系统所采用的调度方式(算法)不完全一样，其中有一种方式就是"分时"原理，或称为时间片轮换法。它所产生的最终效果是使多个正在运行的程序(进程)看上去像是在同时运行一样。因为系统中只有一个 CPU(这里不考虑多处理器系统的情况)，因此这种"同时"只是一种假象，在给定的一个小时间段里，真正能够运行的只有一个进程。

那么，这种"假象"是如何获得的呢？答案是：让所有的进程轮换着进入 CPU 运行，每个进程轮流占用处理器一段很小的时间(如图 3-33 所示)，时间到了就退出并等待，直到下一次轮到再继续。

例如，假设有 A、B、C、D 4 个进程，规定每个进程占用 CPU(资源)运行 20ms。进程 A 首先运行 20ms，不论是否运行结束，都必须让出 CPU 给 B，此时 A 就处于运行挂起[①](就绪)状态；B 占有 CPU 运行 20ms 后转为挂起，将 CPU 让给 C；之后是 D。然后再开始 A 的第二轮……如此循环往复，直到 4 个进程都运行结束为止。

时间片轮换调度算法可以保证就绪队列中的所有进程在给定时间内均能获得一个时间片的 CPU。由于相对于人的感知能力，每个进程所占用的 CPU 时间都很短，使得整个轮换过程进行得非常快，我们无法感觉到进程的"运行""挂起""运行"……就好像每个程序都在同时运行一样(虽然它们实际上平均只有 1/4 的时间在运行)。

操作系统进行进程调度的依据是进程控制块(PCB)。当要调度某个进程执行时，首先从该进程的 PCB 中查出它的现行状态(是就绪、运行还是等待?)和优先级，并以此作为是否执行该进程的调度依据；在调度到某进程后，要根据其 PCB 中所保存的现场信息，恢复其中断前的寄存器、程序计数器等现场信息，并根据程序和数据的地址，找到要继续执行的断点处；在进程执行过程中，当需要和其他进程实现同步、通信或访问文件时，也要依据相应的进程控制信息；当进程因故而暂停执行时，又要在 PCB 中保存中断现场信息；当某进程结束时，则删除其 PCB。

3.5.3 存储器管理

虽然计算机中存储器的容量随着技术的发展一直在不断地扩大，但仍然不能满足现代软件发展的需求，存储器特别是内存依然是一种宝贵而紧张的资源。对其的有效管理，不仅影响到存储器的利用率，也对系统的整体性能有重大的影响。

---

① "挂起"是进程 5 种状态之外的又一种状态。表示暂停的意思，挂起状态也称为静止状态。相应地，非挂起状态就称为活动状态。进程可以由就绪状态转为挂起(静止就绪)，也可以由等待状态或执行状态转为挂起。执行状态转为挂起则进入静止就绪状态。

进程管理解决的是多道程序的并发执行问题，存储器管理解决的是内存的分配、保护和扩充问题。计算机中的作业（处理的数据和程序）首先存放在外存中，在使用（运行）时则需要调入内存（可以考虑一下需要调入内存的原因）。由 3.1.2 节已知，外存和内存在存储容量上相差很大。因此，就引出了这样一些问题：程序是如何调入内存的？将调入到内存的什么地方？如果内存不够用该如何处理？当有多道程序运行时，如何分配内存空间才能最大限度地利用有限的内存为多道程序服务？等等，这些就是操作系统中存储器管理程序要解决的问题。具体包括以下几个方面：

（1）存储分配。为每个作业按一定策略和算法分配所需的内存空间。

（2）地址变换。将程序在外存空间中的逻辑地址转换为在内存空间中的物理地址。

（3）存储保护。保护各类程序（系统的、用户的、应用程序的）及数据区免遭破坏。

（4）存储扩充。解决在小存储空间中运行大程序的问题。

在进一步学习存储管理功能之前，先了解一下程序装入内存的过程。

1. 程序的装入和链接

在多道程序环境中，程序要运行必须要先为之创建进程，而创建进程的第一件事，就是将程序和相应的数据装入内存。

用程序设计语言（无论哪种语言）编写的、完成某项特定的任务的程序称为源程序。源程序编写完成时，都首先存放在外存中，要使其能够被执行，需要装入内存。要将一个用户源程序变为在内存中的可执行程序，通常需要经过以下几步（如图 3-40 所示）：

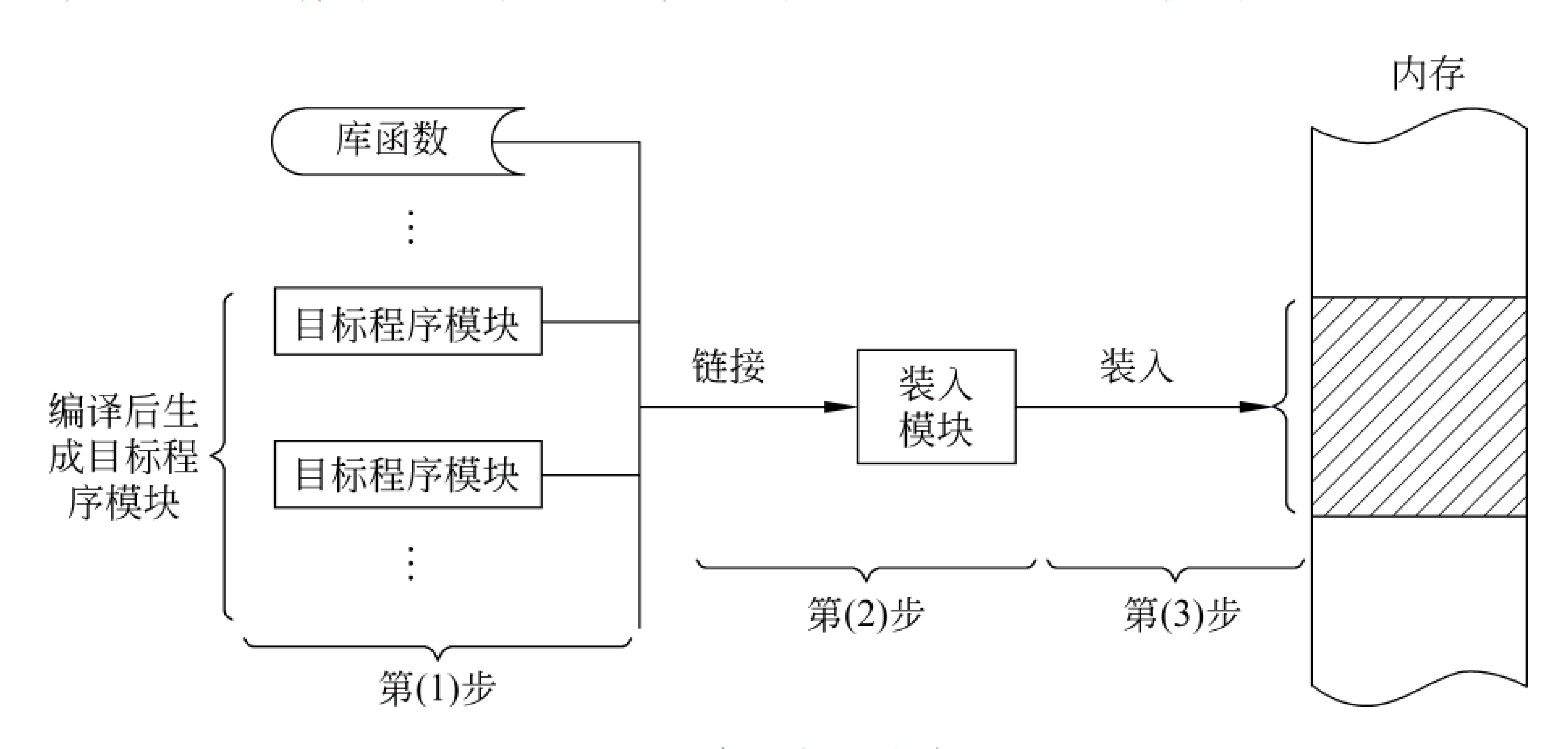


图 3-40　从源程序到装入内存的处理过程

（1）编译（compile）。由编译程序将源程序转换为若干个由二进制机器语言表示的目标（object）程序模块。

（2）链接（link）。将目标程序模块及它们所需要的库函数（已编好的、具有一定功能的、可供其他程序引用的程序段）链接在一起，形成装入模块（load module）。

（3）装入（load）。将装入模块装入内存。

2. 存储分配

在程序中，常常需要明确指令操作的数据或下一条要执行的指令的存放地址（或者说

到哪里去找它们)。编写程序时,这些地址都是用字符或字符串表示的,称为符号地址。源程序经编译后,符号地址被转换为用二进制码表示的相对地址(不是实际运行的地址)。当程序运行要装入内存成为进程时,则要将相对地址转换成主存中的物理地址。CPU 在访问内存时,根据物理地址进行数据或指令的读和写。存储地址的这个变换过程可用图 3-41 表示。

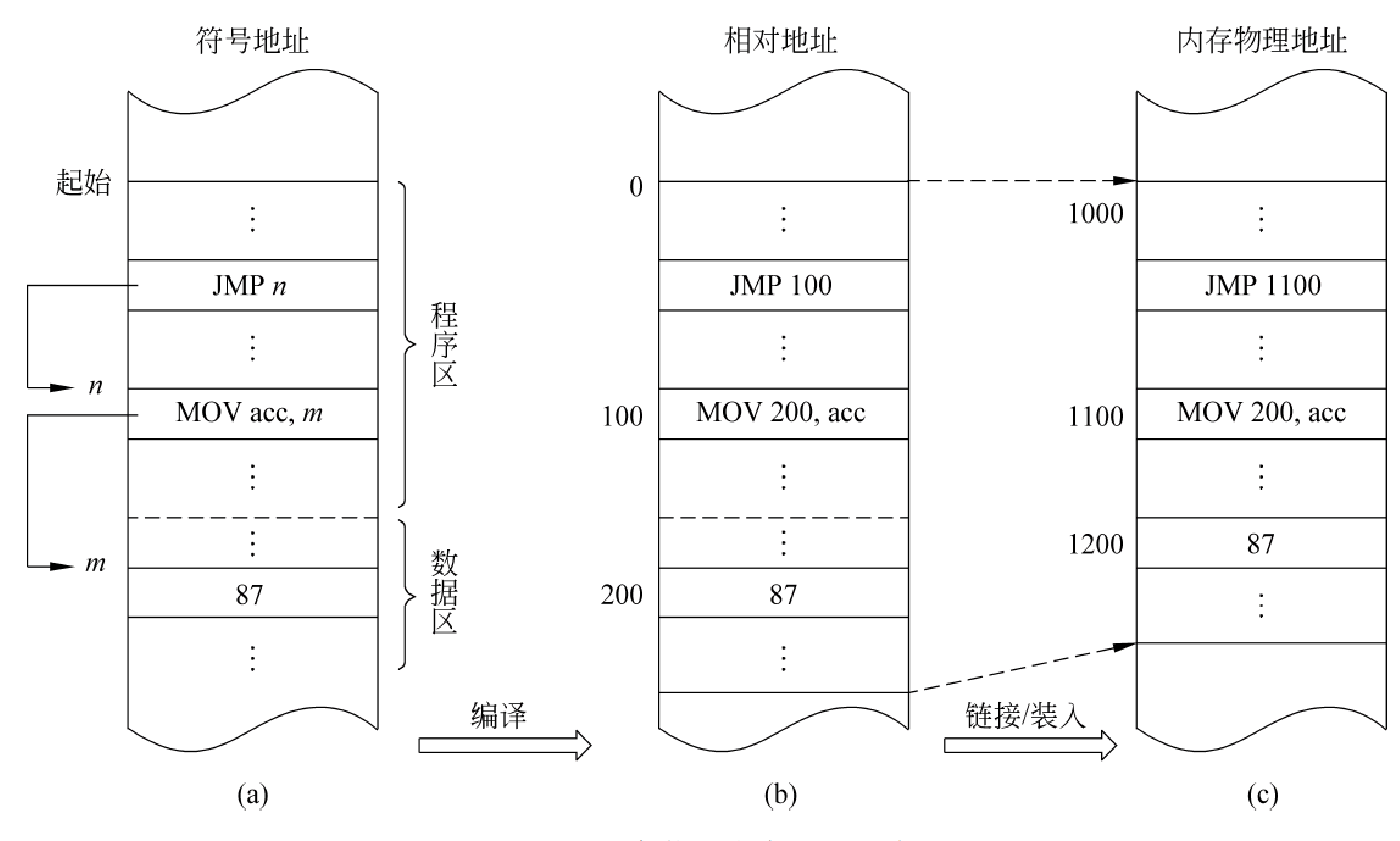


图 3-41 程序装入内存过程示意图

系统在将每道用户作业(程序)从外存调入内存时,就为其创建进程,分配相应的存储空间及必要的资源,并送入就绪队列。将程序装入内存的方式有 3 种:

(1) 绝对装入。根据装入模块中的地址,将程序和数据装入到内存中事先指定的位置。这种方法在程序装入前即要确定所需的内存空间和地址,适合单道程序系统,在多道程序系统中难于实现。

(2) 可重定位装入。由装入程序根据内存当时的实际情况,将程序装入到内存适当的地方,并将相对地址转换为绝对地址(如图 3-41(c)所示)。该方法可用于多道程序系统,但它不允许程序在内存中移动位置。由于一个进程可能会被多次换入或换出内存,每次换入后的位置不一定相同,此时就必须采用动态装入方法。

(3) 动态装入。根据内存的实际情况将程序装入到适当的地方,但要到真正运行时才进行相对地址到绝对地址的转换。现代操作系统多采用动态重定位装入法。

程序装入内存时成为进程,当进程运行完毕后,系统需要将其所占用的内存空间收回,以便装入下一道程序。

3. 存储保护

在计算机中运行的系统程序、应用程序和用户程序都存放在主存中。为了确保各类程序在各自的存储区内独立运行,互不干扰,系统必须提供安全保护功能。措施之一就是

把各类程序的实际使用区域分隔开,使得各类程序之间不可能发生有意或无意的损害行为。这种分隔是靠硬件实现的。用户程序只能使用用户区域的存储空间,而系统程序则使用系统区域的存储空间。

4. 存储扩充

程序只有装入内存才能运行。对一个大程序来讲,若要将其全部装入,则需要较大的内存空间。另外,在多道程序系统中,可能有大量的程序要求运行,但由于内存空间不足以容纳所有希望运行的程序,只能将大量的程序留在外存中。程序从外存装入内存需要一定的时间(这个时间相对于 CPU 来说可是很长的),因此,内存容量不足将直接影响系统的整体性能。

对这一问题,显而易见的解决方法是增大物理内存空间,但这将增大系统成本。另一种方法就是在逻辑上扩充内存容量。"存储扩充"就是解决如何在逻辑上扩充内存容量问题的,即虚拟存储技术。

虚拟存储器由内存和部分硬磁盘组成(如图 3-42 所示),目的是为了克服内存容量的局限性,实现"在小内存中运行大程序"。它力求将外存空间作为内存使用,在逻辑上实现内存空间的扩充,使用户在编程时只考虑逻辑地址空间,而不考虑实际内存的大小。

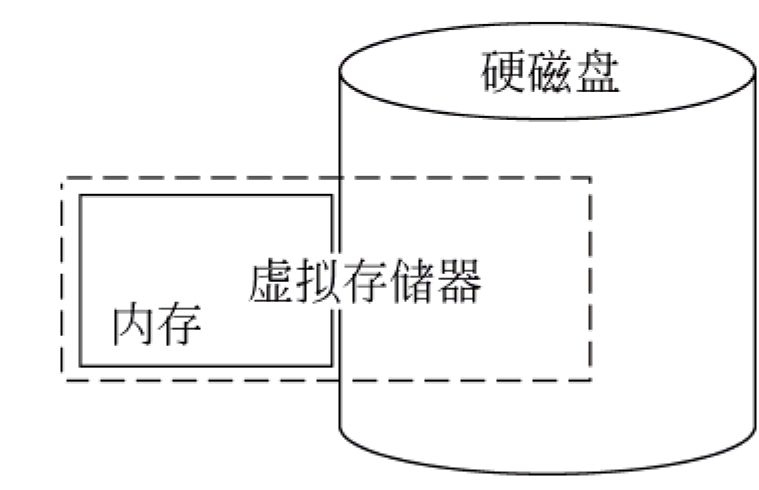


图 3-42 虚拟存储器示意图

程序在虚拟存储器环境中运行时,并不是一次把全部程序都装入到内存,而是只将那些当前要运行的程序段装入内存,其余部分则存留在外存中。如果在程序的执行过程中所要访问的程序段尚未装入内存,则向操作系统发出请求,将它们调入到内存。如果此时内存已满,无法再装入新的程序段,则请求操作系统进行置换,将内存中暂时不用的程序段置换到外存去,腾出足够大的内存空间后,再将所要访问的程序段调入,使程序能够继续运行。

*3.5.4 文件管理

文件是数据的一种组织形式,它可以是计算机处理的数据、图像、文本、程序等各类信息。由于内存的容量有限,且所存信息在断电后即丢失,因此现代计算机系统中大量的程序和数据都是以文件的形式存放在外存中,需要时再调入内存。

如果由用户自己去管理外存上的文件,不仅需要了解文件在磁盘中是如何保存、如何提取、如何删除等的物理细节,而且在多用户环境下,要保证各用户在外存上存放的程序和数据不发生冲突,用户文件不被其他用户窃取和破坏,以及允许在一定条件下的多用户共享文件。显然,一般用户是很难胜任这些工作的,需要靠操作系统的文件管理程序来解决。

因此,文件管理的功能就是有效地管理文件的存储空间,合理地组织和管理文件,并为文件访问和文件保护提供更有效的方法及手段。具体有以下几个方面:

(1) 解决如何组织和管理文件。若要创建一个新文件,用户只要指定文件类型和输

入文件名(路径)即可。具体创建文件、分配存储空间等工作都是文件系统自动完成的。为了分配文件存储空间,文件系统首先对磁盘中的存储块进行管理,包括建立空闲存储块表、对可用存储块进行分配以及回收不用的存储块等。

(2) 实现文件的“按名存取”操作机制。用户按文件名进行操作,系统则是对文件实体进行操作,由文件系统自动完成由文件名到文件实体的对应操作。

(3) 提供文件共享功能及保护措施。

(4) 实现用户要求的各种操作,包括文件的创建、修改、复制、删除等。

1. 文件的组织结构

计算机中存放着成千上万的文件,用户在使用计算机时,可以说随时都在对文件进行操作。用户通常看到的文件是按照目录来组织的,从最上层的根目录(如 C 盘盘符)到下一层子目录(文件夹),再到更下一层子目录,直到最底层的文件。这种提供给用户看的组织结构称为文件的逻辑结构(这是我们都比较熟悉的一种结构),而文件在存储设备上的存放形式称为文件的物理结构。

1) 文件的逻辑结构

文件的逻辑结构是用户所观察到的文件组织形式,与物理存储介质无关。文件的逻辑结构分为有结构文件和无结构文件两种。在有结构文件中,文件由若干个相关记录组成(也称记录式文件,如图 3-43 所示)。记录式文件由一条以上的记录(图 3-43 中的一行)构成,每个记录都包含若干的数据项(图 3-43 中的一列)。数据库文件就属于有结构文件。

SNO	SNAME	SEX	SCLASS	BDATE	RESUME
97000001	王小燕	男	软件971	1978-12-1	班长
97000002	刘丽华	女	软件972	1977-1-15	
97000003	秦刚	男	硬件971	1975-11-30	
97000004	李建国	男	硬件971	1976-6-24	
97000005	郝易平	女	软件971	1977-5-17	
97000006	杨双军	男	软件972	1978-4-28	
97000007	张清效	男	软件971	1978-1-23	副班长

7 共有记录数: 10

图 3-43 记录式文件示意图

无结构文件可以看作是一个字符流(也称流式文件),它由字符序列集合组成,其长度以字节为单位,例如一个源程序文件。可以把流式文件看作是记录式文件的一个特例。

文件系统设计的关键,就在于如何将记录构成一个文件以及如何将一个文件存储到外存中,以便于系统对文件的快速检索。

2) 文件的物理结构

文件的物理结构也称文件的存储结构,是指文件在外存中的存储组织形式。计算机中最常用、最基本的外存是硬磁盘。所以文件的物理结构主要就是研究文件在硬磁盘上的存放和组织方式。通常将外存划分为若干个大小相等的物理块,相应地也将文件划分为相同大小的逻辑块,以块为单位进行存储和管理。需要指出的是,在逻辑上连续的各文件块,在外存上却不一定能存放在连续的物理块中。

3）文件的目录结构

用户使用的是文件的逻辑结构，系统使用的是文件的物理结构，将这两种不同组织结构连接在一起的纽带就是文件的目录结构。

文件目录具有将文件名转换为该文件在外存的物理位置的功能。设计文件按目录组织的主要目的是实现对文件的“按名存取”，并提高文件的检索速度。

为了能对一个文件进行各种操作，文件系统为每个文件设置了一个包含该文件各种属性信息的数据结构，称为文件控制块(File Control Block，FCB)。文件和文件控制块一一对应，文件控制块的有序集合就称为文件目录，每个 FCB 就是一个文件目录项，文件目录本身也是文件，可以称为目录文件(在 Windows 资源管理器中看到的目录文件就是一个个 FCB)。

文件的目录结构有单级结构、二级结构和多级结构。现代操作系统中都采用多级树形目录结构(如图 3-44 所示)，目录的第一级称为根目录，目录树中的非叶结点均为子目录，树叶结点均为文件。



图 3-44　资源管理器窗口中显示的树形目录结构

2. 文件存储空间管理

存储空间管理的主要任务是为文件在外存中分配必要的存储空间，并合理地组织文件的存取方式。

我们已经知道，硬磁盘是按磁道和扇区组织的。为了有效管理磁盘存储空间，通常将磁盘划分为大小相等的物理块，以物理块作为存储分配的基本单位。磁盘物理块以 2^n 个逻辑扇区(簇)构建(磁盘容量不同，n 的取值不同)。例如，将 1KB 或 512B 设为 1 个物理块。

常用的外存分配方法有以下几种：

(1) 连续分配。将每一个逻辑文件中的记录顺序存储到邻接的各物理盘块中，即为每一个文件分配一组相邻的物理块，使文件中记录的逻辑顺序与其在存储器上的物理顺序一致。采用连续存放方式时，文件的访问速度比较快。缺点主要是需要连续的存储空

间，这对大文件是比较困难的。

（2）链接分配。一个文件信息存放在非连续的物理块中[①]，各块之间通过指针连接，前一个物理块指向下一个物理块。为了能有效地管理和组织文件，操作系统维护一个显示所有文件在硬盘中的起始扇区信息的表（当然，这张表本身也在硬盘中），找到了文件存放的起始扇区（文件第一块的存放处），就可以通过“链指针”的方式找到文件第二块的存放地址，依次进行下去，直到找到最后一块。这样，文件的所有物理块就被串联成一个链表，块之间通过指针链接（如图3-45所示）。这里所谓的“链指针”，就是在每个文件块的末尾处给出下一文件块的扇区地址。

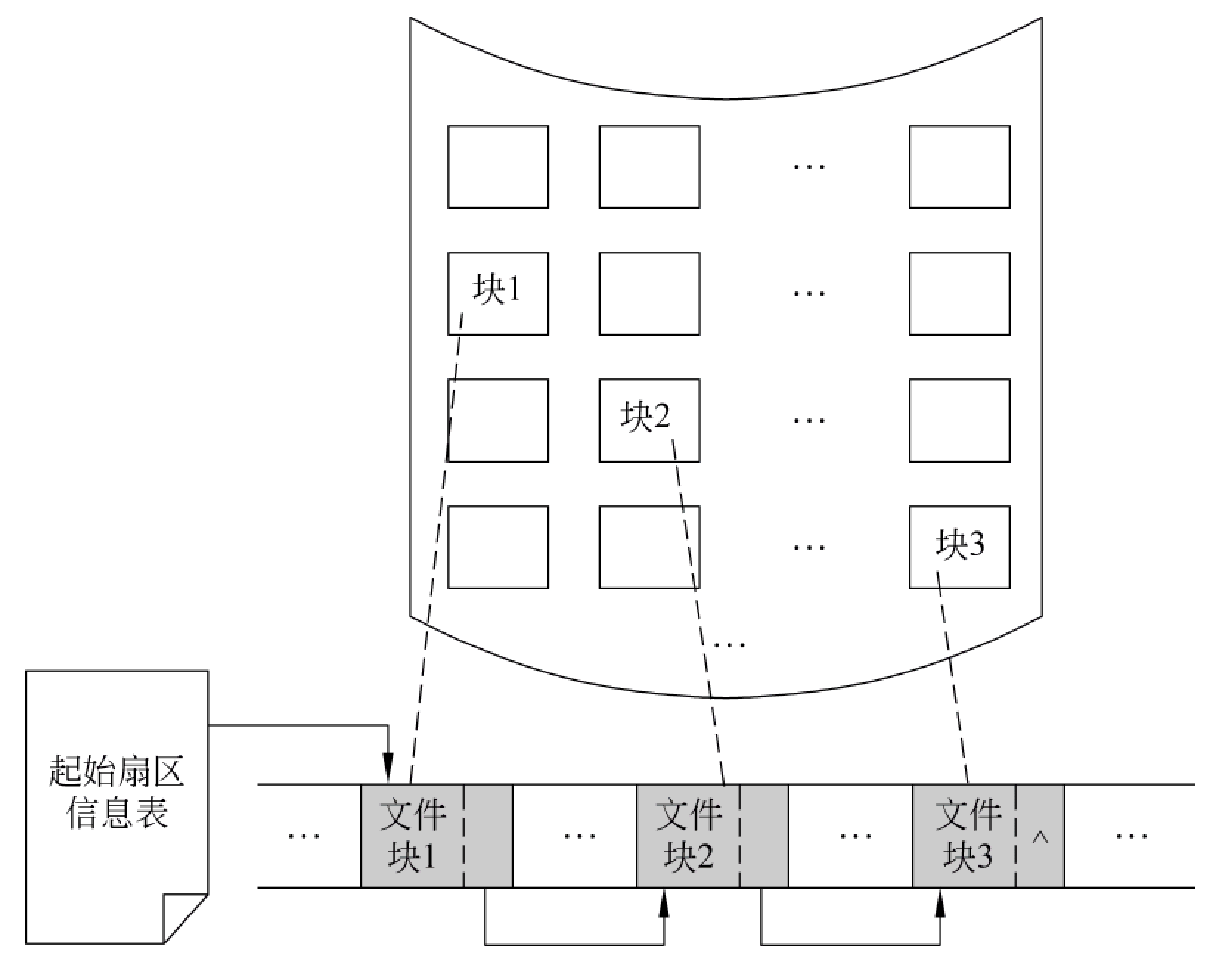


图3-45　链接分配示意图

（3）索引分配。一个文件信息存放在非连续的物理块中。系统为每一个文件创建一张索引表，索引表的表项给出文件的逻辑块号和物理块号的对应关系。

除了为新创建的文件分配存储空间之外，操作系统还需要记住空闲存储空间（未用扇区）的情况。当用户要新建一个文件时，操作系统会核查空闲盘块表，找到一个可以安放该文件的地方。如果用户要删除一个文件，操作系统会同时更新文件在硬盘中的起始扇区信息表及空闲盘块表。首先删除第一张表中的该文件的入口信息，然后将这个文件腾出来的空间信息写入第二张表中。

文件的创建和删除动作贯穿于计算机的整个使用过程，这可能会造成空闲扇区零零散散地散布在磁盘的不同角落（称为磁盘碎片），这会影响存储器的工作性能，特别是在数据查找时，会造成查询时间过长。因此，现代操作系统中都有磁盘碎片整理工具，可以将磁盘中的文件重新进行排列，使得磁盘上每个独立文件中文件块的存放扇区能够彼此尽

① 一个文件在磁盘中存放时，不一定都能够存放于一块连续的物理区域中。当一个文件大到不能在磁盘的一个物理块中放下时，就需要分成若干个“文件块”存放在不同的物理块中，而这些物理块可能是分散地处于磁盘中的不同地方。

可能地靠近，以提高磁盘的读写性能。

3. 文件共享与保护

文件系统中存放着众多的文件，使人联想到以下问题：如何对文件进行保护，免受无意或恶意的破坏？一个文件如何为多用户共享？这些都涉及对访问文件的用户如何进行有效控制的问题。

文件共享是指允许多个用户或程序同时使用一个文件。一个文件能被多个用户共享最简单的方法就是凡需要使用该文件的用户都要自备该文件的副本，显然这会造成存储空间的极大浪费。无论是早期的文件共享方法还是现代文件共享方法，都是解决在一个文件副本的情况下多用户共享的技术和方法。不同的是，共享的范围不断扩大，从单机系统、多机系统、局域网系统到现在的互联网范围中的文件共享。

文件保护实际上有两层含义：文件保护和文件保密。文件保护是指避免因有意或无意的误操作使文件受到破坏，文件保密是指未经授权不能访问文件。这两个问题都涉及用户对文件的访问权限控制。

常见的文件访问控制方式有以下几种：

(1) 访问控制矩阵。系统通过设置一个二维矩阵进行访问控制。二维矩阵的一维是所有用户，另一维是所有文件，对应的矩阵元素则是用户对文件的访问控制权，包括读(R)、写(W)和执行(X)权。当用户向文件系统提出访问请求时，由访问控制验证程序根据矩阵内容对本次访问请求进行比较，如果不匹配则拒绝执行。

(2) 访问控制表。这种方法以文件为单位把用户划分成若干组，同时规定每组的访问权限。这样，所有用户组对文件访问权限的集合就形成了该文件的访问控制表。

(3) 用户权限表。该方法是将一个用户或用户组所要访问的全部文件名集中存放在一个表中，且每个表项指明对相应文件的访问权限。

(4) 口令。口令分两种：一种是系统使用权口令，用于验证是否有权进入系统；另一种是文件使用权口令，当任何用户想使用该文件时，都要核准口令后，才能访问操作。

(5) 密码。密码方式在用户创建源文件并将数据写入存储设备时对文件进行编码加密，在读出文件时对其进行译码解密。只有能够进行译码解密的用户才能正确读出被加密的文件，从而起到文件保密的作用。

*3.5.5 其他功能

1. I/O 设备管理

I/O 设备管理主要是对计算机系统中的输入输出设备进行分配、回收、调度和控制。其主要任务就是负责控制和操纵所有 I/O 设备，实现不同类型的 I/O 设备之间、I/O 设备与 CPU 之间、I/O 设备与通道之间和 I/O 设备与控制器之间的数据传输，使它们能协调地工作，为用户提供高效、便捷的 I/O 操作服务。

操作系统控制着系统中的所有基本 I/O 设备，禁止用户和应用程序直接处理这些

I/O 设备。比如要读取磁盘中的某个文件，不需要给出文件在磁盘上的物理存放地址，而只需要简单地点击鼠标或是调用一个函数，具体到磁盘上去找文件的工作就由操作系统完成了。

2. 用户接口

为了方便用户使用操作系统，操作系统为计算机硬件和用户之间提供了交流的界面。用户通过操作系统告诉计算机执行什么操作，计算机系统为用户提供执行各种操作的服务，并按用户需要的形式返回操作结果。用户和计算机之间的这种交流构成完整的、人机一体的系统，这个系统称为用户接口(interface)。

随着操作系统功能不断地扩充和完善，用户接口更加人性化，呈现出更加友好的特性。目前，人机之间的用户接口有两种主要类型：直接用户接口和间接用户接口。直接用户接口通过交互方式的用户界面进行人机对话，间接用户接口通过程序命令进行系统调用的方式完成人机交流，如图 3-46 所示。

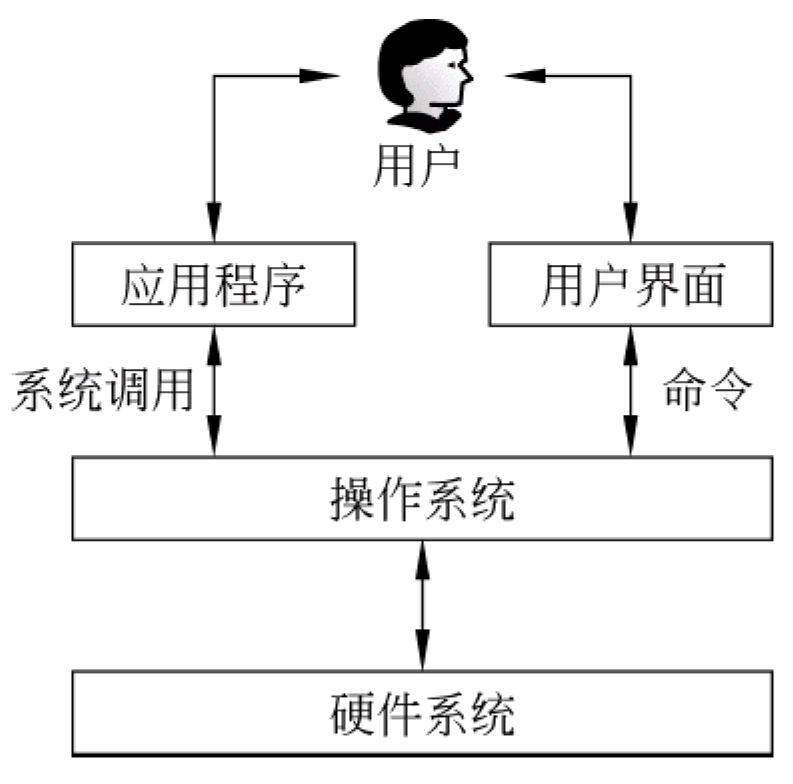


图 3-46 用户接口方式示意图

在计算机系统中，用户不能直接管理系统资源，所有资源的管理都是由操作系统统一负责的。但是，这并不是说用户就不能使用系统资源了，实际上用户可以通过系统调用的方式使用系统资源。这种在程序中实现的系统资源的使用方式被称为系统调用，或者称为应用编程接口(API)。目前的操作系统都提供了功能丰富的系统调用功能。

不同操作系统所提供的系统调用功能有所不同。常见的系统调用分类如下：

- 文件管理：包括对文件的打开、读写、创建、复制、删除等操作。
- 进程管理：包括进程的创建、执行、等待、调度、撤销等操作。
- 设备管理：用于请求、启动、分配、运行、释放各种设备的操作。
- 进程通信：用来在进程之间传递消息或信号等操作。
- 存储管理：包括存储的分配、释放、存储空间的管理等操作。

3. 系统启动

当用户希望运行一个应用程序时，操作系统会将它装入内存。但操作系统本身是如何被装入内存并且执行的呢？处理上述过程的术语称为 bootup(自举)。

计算机启动时，CPU 会使其内部的程序计数器(PC)指针指向一个特殊的位置，例如，Intel 处理器会使 PC 指针指向内存(通常为 ROM)中地址为 0xFFFFFF0H 的地方。自举程序 boot loader 就存放在该地址为首的区域里。所以，一旦计算机启动，首先运行的就是 boot loader 程序。

boot loader 程序的目的是将操作系统从磁盘中装入内存中，一种比较简单的方式是，boot loader 程序会读取磁盘中的一段特定的区域，将该区域中的内容(就是操作系统)复制到内存中，然后在 boot loader 程序的最后执行一条“无条件转移”指令，转移到该

内存地址，操作系统便开始运行了。

一个典型的操作系统（如 Windows）会定义磁盘分区。例如，假设一块磁盘有 1000 个柱面，可以将前 200 个柱面作为一个分区，中间的 500 个柱面作为第二个分区，剩下的 300 个柱面作为第三个分区。这些分区信息汇总为一张表，称为分区表，存放在物理磁盘的第一个块（Master Boot Record，MBR）中。

Intel 处理器将 boot loader 程序包含在 BIOS 中。当计算机启动时，BIOS 中的 boot loader 程序会读取分区表，将操作系统装入内存，然后无条件跳转到操作系统第一条指令的存放处，开始运行操作系统。

习　　题

一、填空题

1. 判断以下语句是否为命题，并说明它们的真值是“真”或“假”。
   (1) 5 能被 2 整除。
   (2) 这朵花真好看啊！
   (3) 4 是偶数。
   (4) 地球绕着太阳转。
   (5) 全体起立！
2. 请将以下复合命题符号化，并说明它们之间的逻辑关系。
   (1) 小明不仅聪明，而且用功。
   (2) 2 不是奇数。
   (3) 今天放假，小明会去看电影或是去打球。
   (4) 5 不能被 3 整除，也不能被 2 整除。
3. 完成下列二进制数的逻辑运算：
   (1) $10110110 \wedge 11010110 =$（　　）
   (2) $01011001B \vee 10010110 =$（　　）
   (3) $\overline{11010101} =$（　　）
   (4) $11110111B \oplus 10001000 =$（　　）
4. 若“与”门的 3 位输入信号分别为 1、0、1，则该“与”门的输出信号状态为（　　）。若将这 3 位信号连接到“或”门，那么“或”门的输出又是什么状态？
5. 在图 3-47 中，要使 $Y=0$，$A_0 \sim A_3$ 的状态必须为
   (a)（　　）；(b)（　　）；(c)（　　）；(d)（　　）。
6. 计算机硬件能够直接识别的指令是（　　）。
7. 冯·诺依曼计算机的基本原理是（　　）。
8. 冯·诺依曼结构的主要特点是（　　）、（　　）和（　　）。
9. 与冯·诺依曼结构相比，哈佛结构主要具有（　　）和（　　）两大特点。

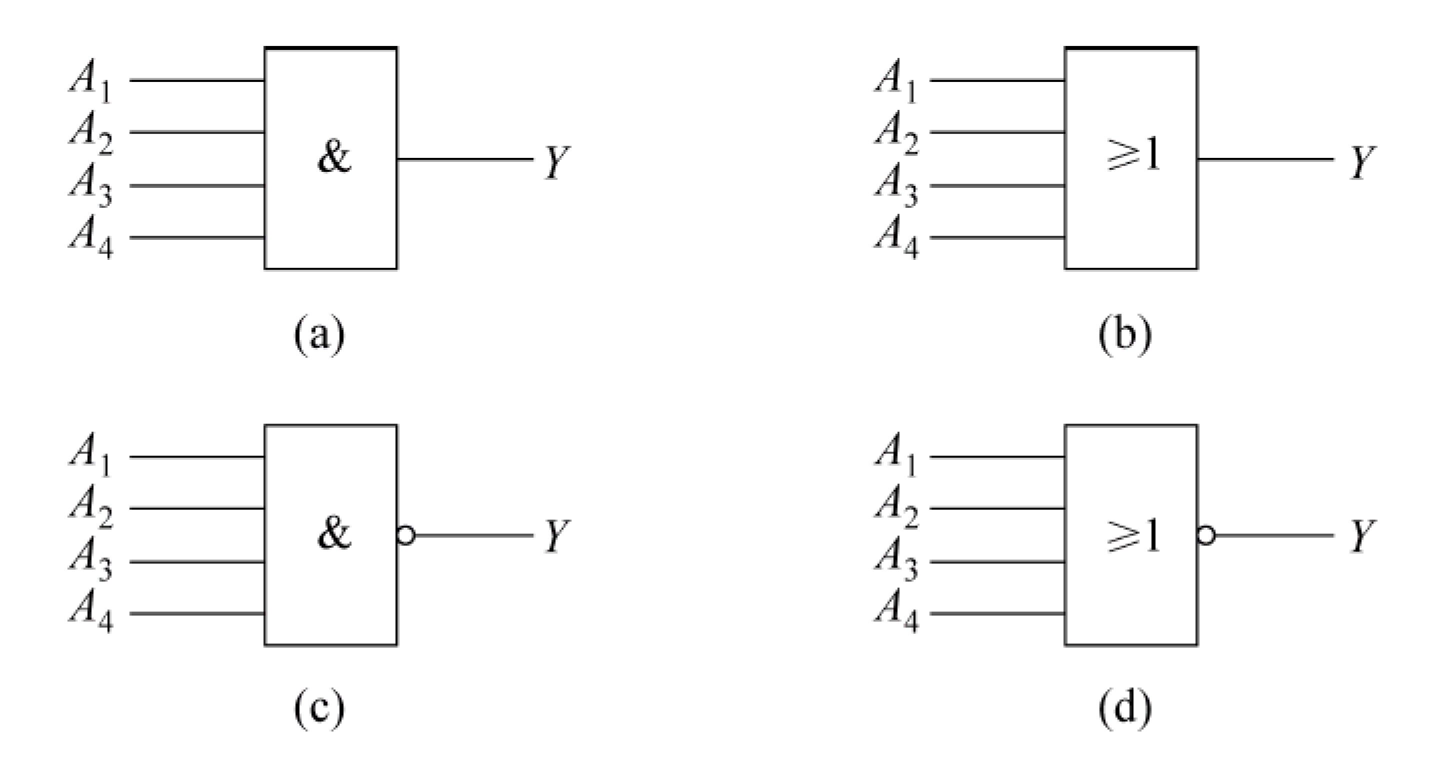


图 3-47　题 5 图

10. 某程序段经编译后生成 98 000 条机器指令，假设取指令、分析指令和执行指令所用的时间均为 2ns，则使用并行流水线方式完成该程序段所需的时间为(　　)ns。
11. 如果说图灵机 A 能够完全模拟图灵机 B，则意味着(　　)。如果 A 和 B 能够相互模拟，则表示(　　)。
12. 操作系统的基本功能包括(　　)、(　　)、(　　)、(　　)和用户接口。
13. 进程在其生命周期中的 3 种基本状态是(　　)、(　　)和(　　)。
14. 数据库中的文件类型属于(　　)文件。
15. 程序装入内存时，源程序中的符号地址最终要变换为内存的(　　)地址。

二、简答题

1. 根据式(3.8)，练习用“与”门、“或”门、“非”门构造出“异或”门。
2. 试说明指令的执行步骤，哪些步骤是必需的？
3. 简述冯·诺依曼计算机的特点。
4. 简述进程和程序的区别。
5. 说明为什么要引入进程。
6. 将程序装入内存必须经过哪些步骤？



第 4 章　计算机网络及应用

引言

计算机网络是信息传输的基础设施,与信息获取、信息交换和信息发布密切相关。本章首先介绍计算机网络的基础知识,然后介绍因特网的基本概念和应用,最后简要描述网络安全的概念、网络安全技术、常见的网络威胁及其防护技术。

教学目的

- 了解计算机网络的基本概念、分类和应用方式。
- 理解网络协议和网络体系结构。
- 了解因特网中 MAC 地址、IP 地址、域名等概念。
- 了解因特网接入技术。
- 了解常见的因特网的应用及其相关技术。
- 了解计算机网络安全及相关技术。
- 了解常见的网络威胁及其防护技术。

4.1　计算机网络基础知识

计算机网络是计算机技术与通信技术相结合的产物。如今,计算机网络在全球范围内得到了广泛的普及,日益深入到国民经济各部门和社会生活的各个方面,成为当今信息社会的重要基础和人们日常生活工作中不可缺少的工具。

4.1.1　概述

1. 计算机网络的概念

计算机网络(computer network)是指将一些位于不同地理位置的、自治的计算机通

过通信线路连接起来,实现资源共享和信息传输的计算机系统(如图 4-1 所示)。这里所谓的“自治的计算机”是指不依赖其他计算机而能够独立运行的“智能”系统,如微型计算机、智能手机等。计算机网络中的计算机又称为主机(host)。

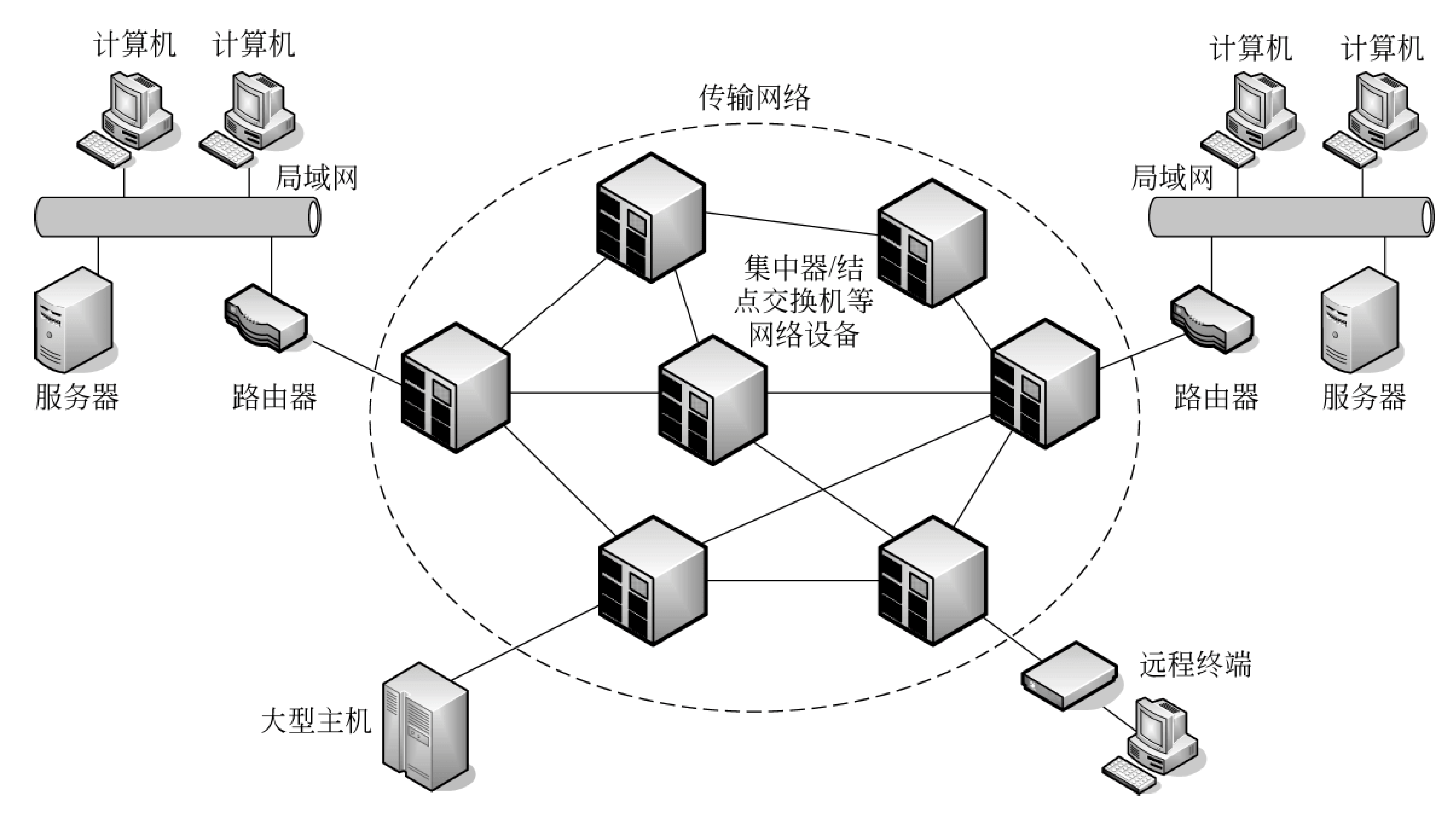


图 4-1　现代计算机网络示意图

基于计算机网络的资源共享主要包括以下几种类型的资源:

(1) 硬件资源,如大容量存储设备、网络打印机、绘图仪等大型设备。

(2) 计算资源,超级计算机系统提供的大数据量、高速运算能力。

(3) 软件资源,各种软件(包括共享软件和购买了许可证的收费软件)。

(4) 数据资源,如各种统计数据、文献数据库、音视频、资料等。

在这些资源中,数据资源是最为重要的,因为其他几种资源可以通过购买而获得,而数据资源在很多情况下难以通过经济手段获得,特别是用户在工作中积累起来的数据。

最典型的计算机网络实例就是人们日常广泛应用的因特网(Internet)。因特网是世界范围内最大的计算机网络,它由成千上万个局域网(Local Area Network,LAN)和各种主机通过许多路由器(router)互联而成,人们往往称其为“网络的网络”(network of networks)。

在概念上,与计算机网络类似的系统还有分布式计算机系统。分布式计算机系统可以在分布式操作系统支持下进行分布式数据处理和并行计算,也就是说,系统中的各计算机可以互相协调工作,共同完成一项任务,一个大型程序可以分布在多台计算机上并行运行。在分布式系统中,各计算机对用户是透明的。在用户看来,分布式系统就好像一台单独的计算机一样。分布式操作系统为用户选择一台或多台计算机来运行其程序,并将运行的结果合并传送给用户,这些都不需要用户的干预。分布式系统需要计算机网络作为底层通信支持。

2. 计算机网络的发展

计算机网络源于计算机与通信技术的结合，其发展经历了如下几个阶段。

1）以单主机为中心的联机系统

以单主机为中心的联机网络系统被称为第一代网络。20 世纪 60 年代中期以前，计算机主机极为昂贵，而通信线路和通信设备的价格相对便宜，为了利用强大的主机处理能力进行数据处理，人们通过通信线路将多台终端设备与主机连接，这样就构成了以单主机为中心的联机网络系统，如图 4-2 所示。

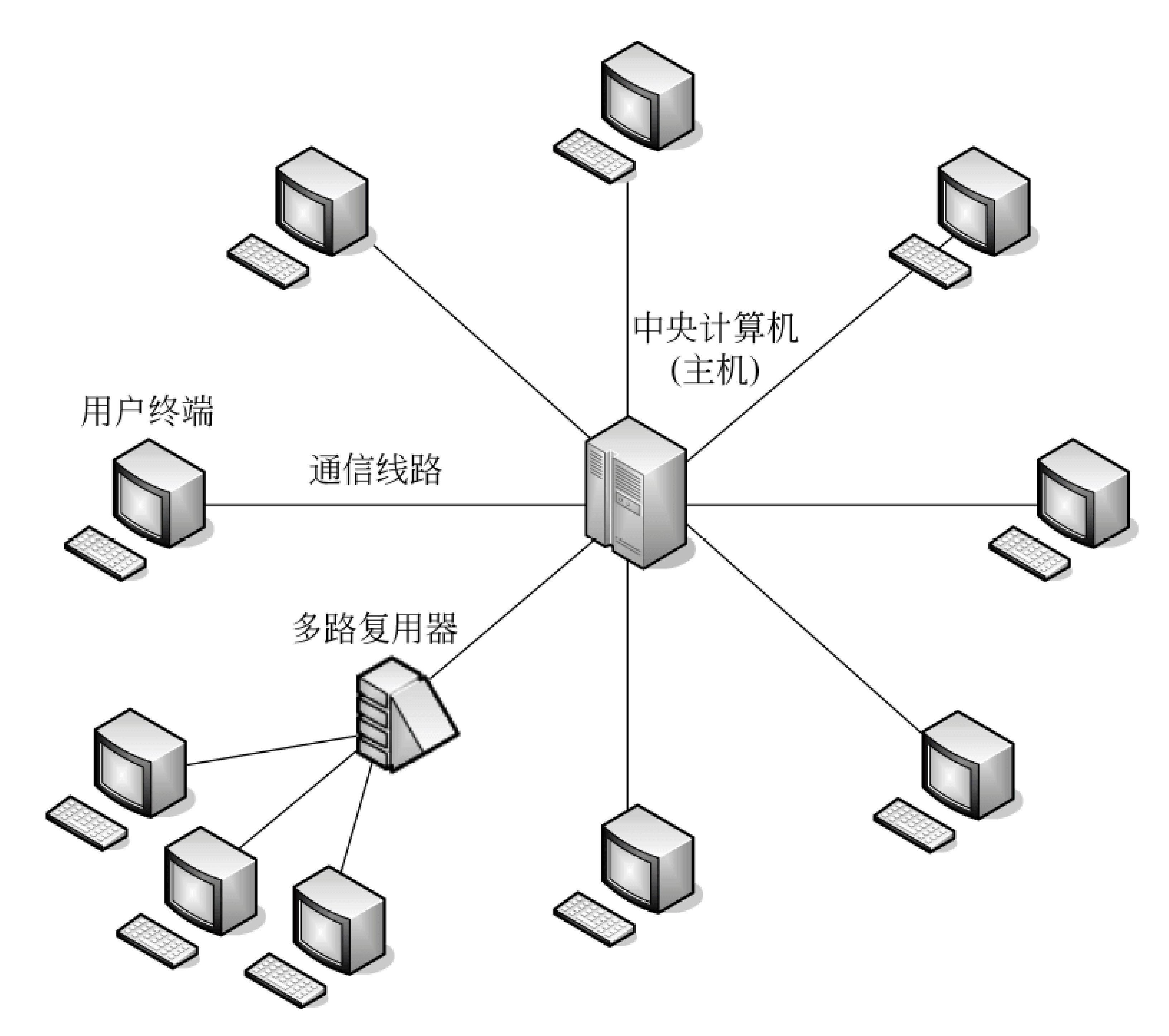


图 4-2　以单主机为中心的联机系统

在联机系统中，终端是没有处理能力的非智能设备（由键盘、显示器和通信接口构成），主机则具有强大的计算和处理能力。工作时，用户通过终端向主机发出操作请求，主机响应终端的请求完成相应的处理，并将结果回送给终端进行显示。

联机网络系统主要有 3 个缺点：一是主机负荷较重，需要承担数据处理和数据通信两方面的任务。二是通信线路的利用率低、成本高，尤其是距离较远时，分散的终端都要单独占用一条通信线路。不过，这个缺点可以通过在终端密集的区域采用终端集中器（或多路复用器）来合并通信流量予以解决。三是联机系统属于集中控制方式，可靠性低，中央主机的失效直接导致整个系统的崩溃。

2）分组交换网络的出现

随着计算机技术和通信技术的进步，从 20 世纪 60 年代中期开始，人们开始研究如何将多个联机系统互相连接，以便在更大范围内实现数据通信和资源共享。研究的结果最终导致了分组交换网络的出现。其间经历了两个演变过程，开始仅仅是简单地将各联机系统的主机通过通信线路互连，如图 4-3(a)所示。后来又将数据通信任务从主机中分离

出来,交由专用的通信处理机(Communication Control Processor,CCP)完成,主机仅完成数据处理任务。CCP 之间通过通信线路互连,完成主机之间的数据通信。由多个 CCP 组成的网络称为通信子网,提供数据传输服务;而主机的集合称为资源子网,提供资源共享服务。两者构成了逻辑上具有两个层次的网络,如图 4-3(b)所示。

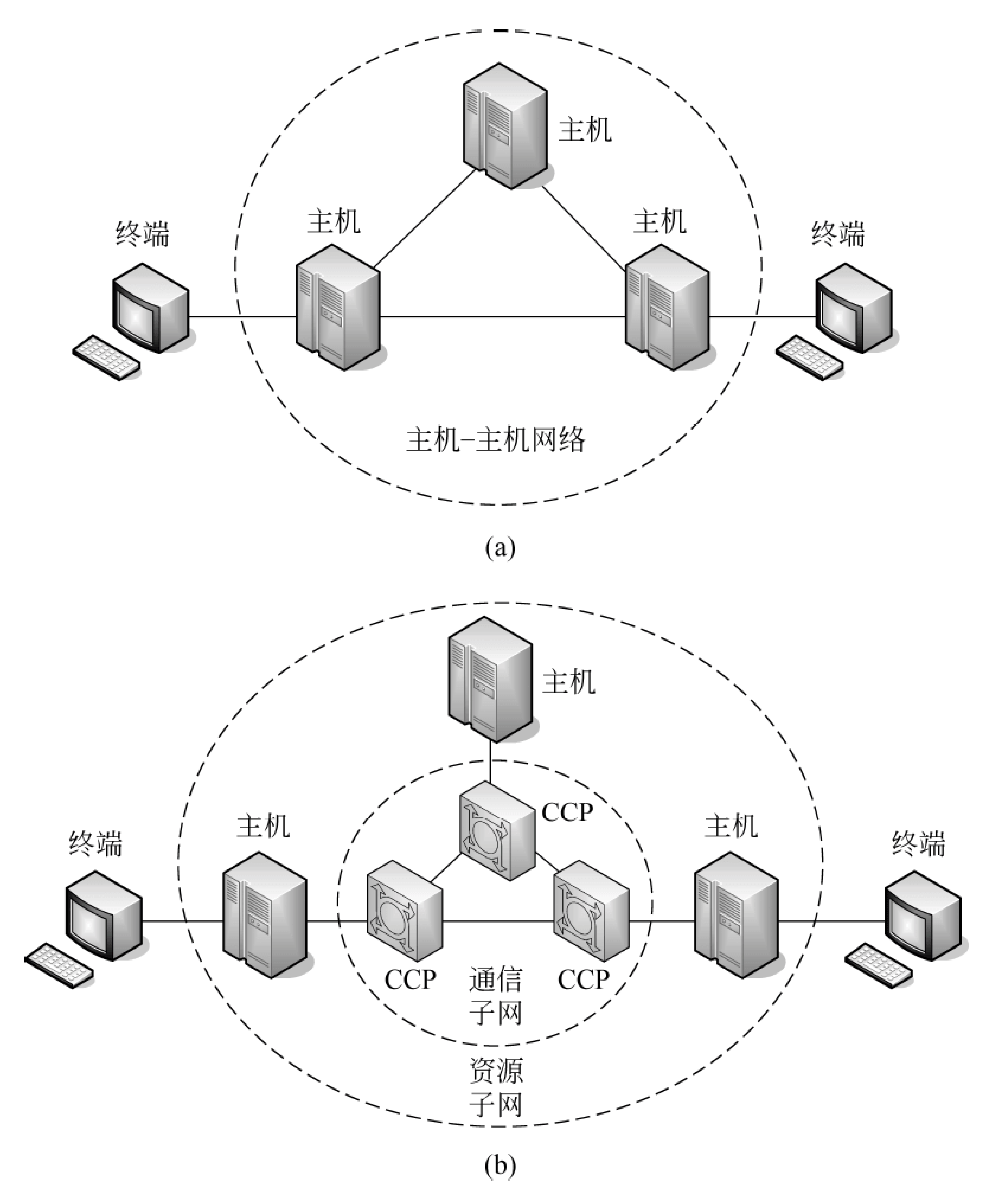


图 4-3　分组交换网的结构演变

分组交换网的另一个特征就是将传输的数据分割成若干个分组进行传输,而不是像原来那样一次全部传输。这个改变是革命性的,它使网络的传输效率和可靠性有了极大的提升,为今天的 TCP/IP 网络打下了基础。

3)网络体系结构标准化时代

20 世纪 70 年代以后,世界出现了大量的计算机网络,它们大都由研究部门、大学或公司各自研制开发,没有统一的体系结构,难以实现互连。于是,开放(open)就成了计算机网络发展的主题。

1977 年国际标准化组织开始研究计算机网络体系结构的标准化问题,经过多年艰苦的努力,于 1983 年推出了开放系统互连参考模型(Open System Interconnection/Reference Model,OSI/RM),简称 OSI 标准。

目前,OSI 标准中所提出的关于计算机网络体系结构的概念已被人们广泛地接受,成

为网络研究和开发的“圣经”。也正是在 OSI 标准的推动和影响下，使得后来的 TCP/IP 体系结构迅速普及。TCP/IP 体系结构虽然不是国际标准，但它的发展、应用和影响力都远远超过了 OSI 标准，并成为现代因特网的标准。

4）因特网时代

20 世纪 90 年代，世界许多国家相继建立了本国的主干网络，并相互连接，逐渐演变成了今天的因特网。因特网采用 TCP/IP 协议标准，是一个由成千上万个网络构成的全球性的互联网。

3. 计算机网络的组成

就像任何智能系统都是由硬件和软件两部分组成的一样，计算机网络同样由网络硬件和网络软件组成。

1）网络硬件

网络硬件主要分为两大类，即网络结点和通信链路。网络结点又分为端结点和转接结点。端结点是指通信的源设备和目的设备，例如具有智能的主机系统和非智能的终端设备。转接结点是指用于在网络中控制和转发信息的中间结点设备，例如各种交换机、集中器、多路复用器、调制解调器、光通信设备等。通信链路是指传输信息的信道，可以是有线的，如同轴电缆、光纤、双绞线等；也可是无线的，如微波、无线电、红外线、卫星等。

2）网络软件

计算机网络是一个智能系统，需要通过软件来控制网络硬件的运行和管理。另外，为了安全有效地利用网络资源，为用户提供资源共享服务和信息发布服务，也需要通过软件对网络资源进行全面的管理、调度和分配，并提供安全防护功能。网络软件是实现网络功能不可缺少的软件环境。网络软件主要包括以下几类：

（1）网络协议软件，实现网络协议功能，通常被嵌入到操作系统中，如 TCP/IP。

（2）网络应用客户端软件，为用户提供使用网络的界面，如 IE 浏览器、Outlook、微信等。

（3）网络操作系统，实现系统资源共享，管理资源访问，提供网络通信的程序集合。流行的网络操作系统主要有 Windows、Linux、UNIX 等。

（4）网络工具软件，实现对网络的监控、管理和维护等功能。

4. 计算机网络分类

计算机网络的分类方法很多，从不同角度对网络进行观察和分类，有利于全面了解网络系统的各种特性。常见的网络分类方法有按覆盖范围划分和按拓扑结构划分。

1）按覆盖范围划分

（1）广域网（Wide Area Network，WAN）。

广域网的覆盖范围从几十千米到几千千米，可以是一个地区或一个国家，甚至是世界范围。广域网通常利用各种公用交换网络，将分布在不同地区的计算机网络或计算机系统互连起来。广域网的特点是：

① 覆盖地理范围大。

② 适应大容量与突发性通信的要求。

③ 适应综合业务服务的要求，提供 QoS 能力。

④ 开放的设备接口与规范化的协议。

⑤ 完善的通信服务与网络管理。

广域网的典型通信速率从 56kb/s 到 622Mb/s。由于距离很远，广域网的传播延迟较大，可从几毫秒到几百毫秒（卫星信道）。

在拓扑结构上，广域网呈现为由许多交换机互连而成的网状结构，交换机之间采用点到点线路连接，采用的传输介质包括光纤、无线电、微波、卫星等。

在体系结构上，广域网工作在 OSI 参考模型的最低 3 层，主要采用存储转发方式进行数据交换。

（2）局域网（Local Area Network，LAN）。

局域网的覆盖范围通常为几米到几十千米，一般部署在一个建筑物内，或在一个工厂、一个企事业单位内部。局域网的特点是：

① 覆盖地域范围小。

② 系统灵活性高，建设、维护、扩展、改造、升级都非常简单和容易。

③ 传输速率高（10Mb/s～10Gb/s）。

④ 传播延迟小，可靠性高。

⑤ 通常由某个单位自行建设、管理和维护。

局域网的拓扑结构常见的是总线和环形，这是其有限的地理范围所决定的。局域网采用的传输介质包括双绞线、光纤和无线。

目前，非常流行的以太网（Ethernet）就是一种典型的局域网。

（3）城域网（Metropolitan Area Network，MAN）。

城域网的覆盖范围在广域网与局域网之间，其运行方式与局域网相似。如果不作严格的区分，城域网可以认为是一种城市或地区范围的局域网，它可以覆盖一组邻近的公司、一个城市或一个地区。城域网一般采用光纤作为传输介质，采用的技术既包括广域网技术，也包括局域网技术，可以支持数据、声音、图像和视频的传输，并有可能涉及当地的有线电视网。

2）按拓扑结构划分

拓扑（topology）是从图论演变而来的，是一种研究与大小形状无关的点、线、面的特点的方法。在计算机网络技术中，抛开网络中的具体设备，把计算机、服务器、交换机、路由器等设备抽象为“点”，把通信介质或通信链路抽象为“线”，这样从拓扑学的观点来观察计算机网络系统，就形成了点和线组成的几何图形，从而抽象出了计算机网络的具体结构。这种采用拓扑学方法抽象的网络结构称为计算机网络的拓扑结构。

计算机网络的拓扑结构主要有总线型、星形、树形、环形、不规则形和全互连型等几种，如图 4-4 所示。网络拓扑结构对网络的设计、传输控制方法、传输技术、可靠性、费用等方面有着重要的影响。

（1）星形结构。

星形结构包括一个中心结点和一些通过点到点链路与中心结点相连的端结点（计算

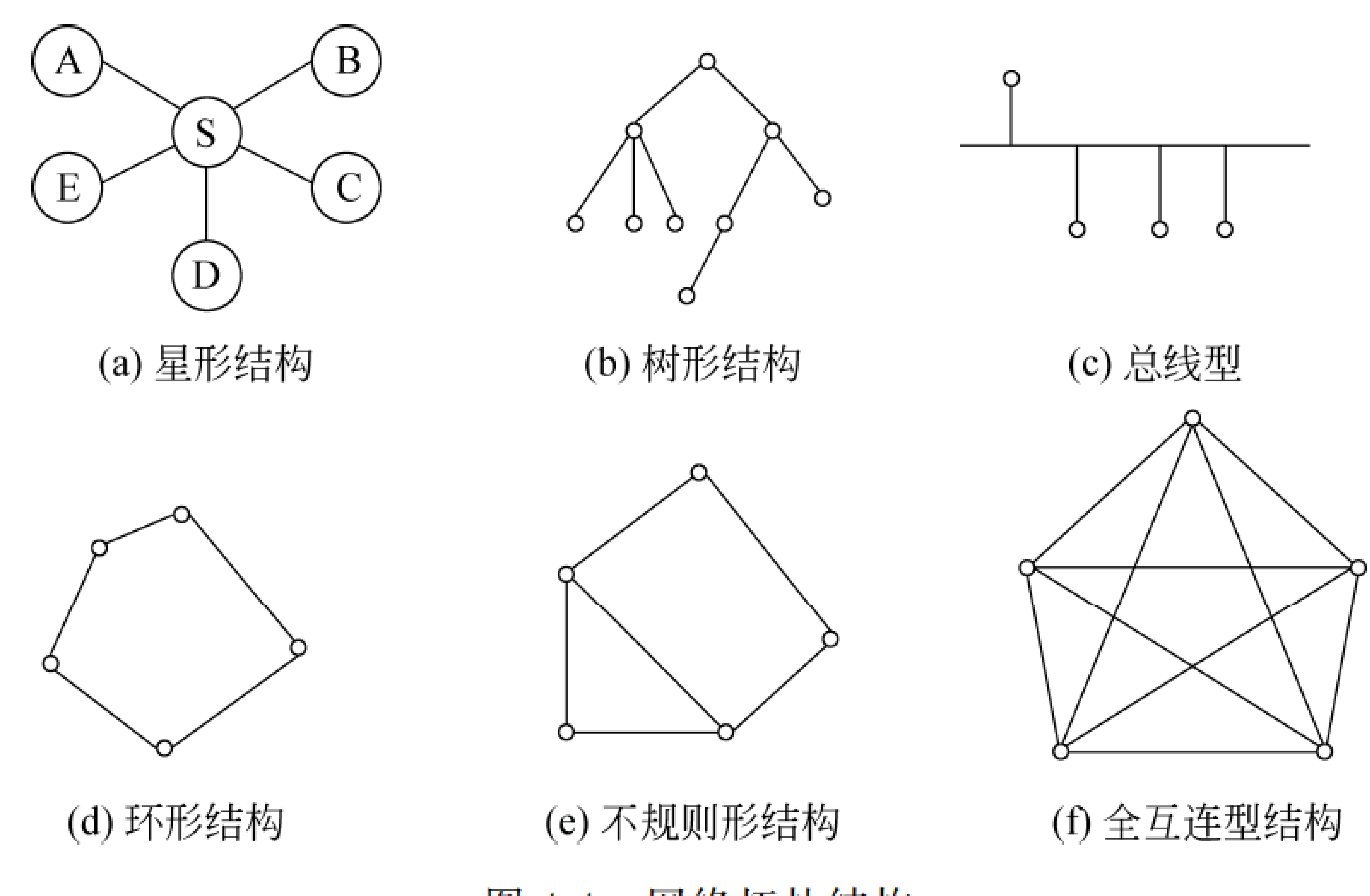


图 4-4　网络拓扑结构

机)，如图 4-4(a)所示。中心结点有两种类型，一类是仅用来连接各从结点的集中连接设备，对传输的数据不进行处理；另一类是具有处理能力的设备，能够对传输的数据进行存储、处理和转发。星形结构的优点是构建容易，易于扩充，控制相对简单。其缺点是采用集中控制，对中心结点的可靠性要求较高。

(2) 树形结构(层次结构)。

网络呈现为一棵倒置的树形结构，树的叶子结点是计算机，根结点和中间结点是集中连接设备，如图 4-4(b)所示。树形结构如果仅有两级，就变为星形结构。一般来说，规模比较大的网络都采用树形结构来构建。

(3) 总线型结构。

总线型结构是由一条公用总线连接若干个端结点所形成的网络，如图 4-4(c)所示。在总线型网络中，数据传输采用广播通信方式，即一个结点发送的信息会被广播到网络上的其他结点。由于总线是公用的，如果有两个以上的结点同时发送数据，就会产生冲突，因此必须采取某种方法来控制总线的有序使用，这就是所谓的介质访问控制方法(Medium Access Control Method)，其典型例子就是以太网中使用的载波监听多路访问/冲突检测(Carrier Sense Multiple Access/Collision Detect，CSMA/CD)。

总线型网络的优点是结构简单、成本低，缺点是结点发送数据时需要争用总线，所以实时性较差，当网络通信量增加时，性能会急剧下降。

(4) 环形结构。

环形结构呈现为一种首尾相连的闭合环，环中的结点以点到点方式两两连接，如图 4-4(d)所示。环形网络常常使用令牌控制方法来协调各结点的数据传输。

(5) 点到点部分连接的不规则形结构(网状结构)。

网络中的结点以点到点方式连接，每个结点至少有一条链路与其他结点连接，如图 4-4(e)所示。这种结构多用于广域网。广域网中各结点间的距离很远，如果每个结点都与其他结点使用点到点线路连接，会造成线路建设成本太大，非常不经济。因此某些结点之间是否使用点到点线路连接，要依据信息流量以及结点的地理位置而定，如果结点间

的通信可由其他结点转发而不影响传输性能时,则可不必直接互连。因此网络覆盖地域范围很大且结点数较多时,往往采用部分结点连接的不规则形拓扑结构。不规则形结构的网络必然会出现经由中间结点转发进行通信的现象,这称为交换。用于转发数据的设备称为交换机。

(6) 点对点全互连结构。

如果网状结构中每个结点和其他所有结点都有连接,这就构成了全互连结构,如图4-4(f)所示。全互连结构的连接数随结点数量的增加而迅速增加。例如,具有10个结点的全互连结构,整个网络共需45条线路。显然,在大型网络中,全互连结构的线路建设成本是非常高的。

4.1.2 网络体系结构和协议

计算机网络是一个复杂系统,涉及了众多的概念、原理、技术和方法。就像软件开发一样,人们在开发一个软件时往往会按照模块化、分层次(自顶向下)的方法来进行程序设计,在处理计算机网络这种复杂系统时也不例外,也采用了"分而治之"的层次化的方法对其进行研究、设计和开发。通过分层可以将一个庞大而复杂的问题转化为多个简单的局部问题,分别处理和解决。

1. 网络体系结构

网络体系结构是用于设计和实现计算机网络的一组原则。它提供了独立于技术途径描述计算机网络的相关概念,其中包括必须由网络执行的功能、信息格式与交互过程的描述等。

如上所说,为了降低研究计算机网络的复杂性,网络体系结构是按层(layer)的方式来组织的,每一层的功能实现都以其下层为基础(换句话说,每一层的实现都要依赖于下层的服务),这些功能具体体现为一组通信协议,各层协议的集合称为协议集(或协议栈)。另外,为了实现各层的独立性,每层在功能上的具体实现细节对上一层屏蔽,层与层之间的耦合仅通过层间接口来实现。这就是计算机网络体系结构的分层思想。

简而言之,一个网络应包括哪些层次,各层具有哪些功能,各层包括哪些协议,层次间如何交换数据,这些都是网络体系结构所要研究的问题。

目前,最典型的计算机网络体系结构有两种:OSI/RM(7层)和TCP/IP(4层)。

OSI/RM规定的7个层次从上到下分别为应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。TCP/IP规定的4个层次从上到下分别为应用层、传输层、网际层(也称互联网层)和网络接口层。

2. 网络协议

在计算机网络中,通信双方要做到有条不紊地交换数据,就必须遵守一些预先约定好的规则,这些为网络中数据交换而建立的规则、标准或约定就是网络协议。

典型的网络协议要包含3个方面的内容(三要素):

(1) 语法：数据与控制信息的组成结构或格式。

(2) 语义：协议中控制信息的含义，如帧的起始定界符、源地址和目的地址、帧校验序列等。语义还规定通信双方对某种事件应该做出的动作及响应，如在什么条件下进行应答，什么条件下进行重发等。

(3) 时序：规定了通信双方的操作顺序。例如，某协议规定：源站发送数据报文，若目的站接收的报文无差错，就向源站发送确认信息，通知源站报文已被正确接收；若目的站发现收到的报文有差错，就丢弃收到的报文；源站在规定时间内如果没有收到确认信息，则重发报文；等等。

下面通过一个简单的例子来使读者对网络协议有一个感性认识。

【例 4-1】 网络中的站点 A 和站点 B 要进行点到点通信，为保证两个站点都能理解对方传送的信息并执行相应的操作，规定双方通信的协议如下。

(1) 帧格式。信息以帧为单位进行组织，帧格式如下：

SYN	STX	Text	ETX	FCS

其中：

SYN 标识一帧的开始，长度为 1B，编码为 16H。

STX 标识正文的开始，长度为 1B，编码为 02H。

Text 为信息正文，长度为 nB，允许的编码为 ASCII 字符集中的 20H～7EH。

ETX 标识正文的结束，长度为 1B，编码为 03H。

FCS 为校验码，长度为 2B，计算方法：从 SYN 到 ETX 所有字节之和(包括 SYN 和 ETX)，高字节在前，低字节在后。

(2) 传输规则。以字节为单位进行发送/接收。

【发送方】

① 按帧格式组帧。

② 将帧从左到右逐字节发送。

③ 接收 ACK，若在 1 秒内没有收到 ACK(06H)，则转②重发此帧。若重发 5 次后还不成功，则报告错误。

④ 结束。

【接收方】

① 检测 SYN 字符，若发现 SYN，则开始接收帧，直到接收到 ETX 为止，然后再接收 2B 的 FCS。

② 按校验码计算方法求出一个 16 位的值，与接收到的 FCS 比较。若相同，说明接收的帧无差错，就发送 ACK(06H)进行确认；若不同，说明接收的帧有错误，就丢弃收到的帧，转①。

③ 提取帧中的信息正文(Text 字段)，保存到接收缓冲区中。

④ 结束。

在这个协议例子中所定义的信息格式就是协议的语法，帧中各语法元素的含义就是

协议的语义，帧的传输规则就是协议的时序。一般情况下，本例中的帧统称为报文(message)，语法元素统称为字段(field)，其中的 Text 字段统称为数据(data)或载荷(load)，Text 以外的其他字段统称为控制信息(control message)。要注意的是，不同协议的语法、语义和时序的差别是非常大的，这取决于通信需求和协议的设计。

3. 协议分层与封装

根据网络体系结构中的分层原则，网络功能是被划分到不同层次实现的。或者说，通信过程中的各种问题在不同层次上予以解决的。

显然，通信双方要能够实现交互，双方同一层次(称为对等层)中的进程必须按照相同的规则(或协议)来与对方通信，这就意味着双方同一层次的协议也应该是相同的。由此可知，网络的每一个对等层都需要有协议来规定双方在这一层次上的数据交换规则，即网络协议也是分层次的。概括地说就是：

(1) 网络体系结构中的每一层都有若干个通信协议，这些协议支配着本层通信双方之间的数据传输。

(2) 以网络体系结构的观点来观察网络中两个站点之间的通信，可以发现它是在多个对等层通信的支持下实现的。但对等层通信只是一种由软件实现的逻辑通信，实际的数据传输最终还是要归结为传输介质上的物理通信。

协议的具体实现形式是数据封装(encapsulation)。所谓数据封装就是在数据前面加上用于控制传输过程的控制信息，如地址、数据长度、校验码等，这些控制信息统称为报文头部(header)。与数据封装对应的操作是数据解封装(decapsulation)，解封装就是从接收的报文中去掉报文头部，取出其中的数据。对等层之间进行通信时，发送方执行数据封装操作，接收方执行解封装操作。这个过程就像日常生活中邮寄信件一样，寄信方将信件放入信封(封装)，邮局根据信封上的“控制信息”传输邮件，收信方将信封拆开取出信件(解封装)。

【例 4-2】 一个具有三层结构的网络中的数据传输过程。

在这个例子中，网络具有三层结构，如图 4-5 所示。假定网络中某一方的用户要发送数据给另一方的用户。

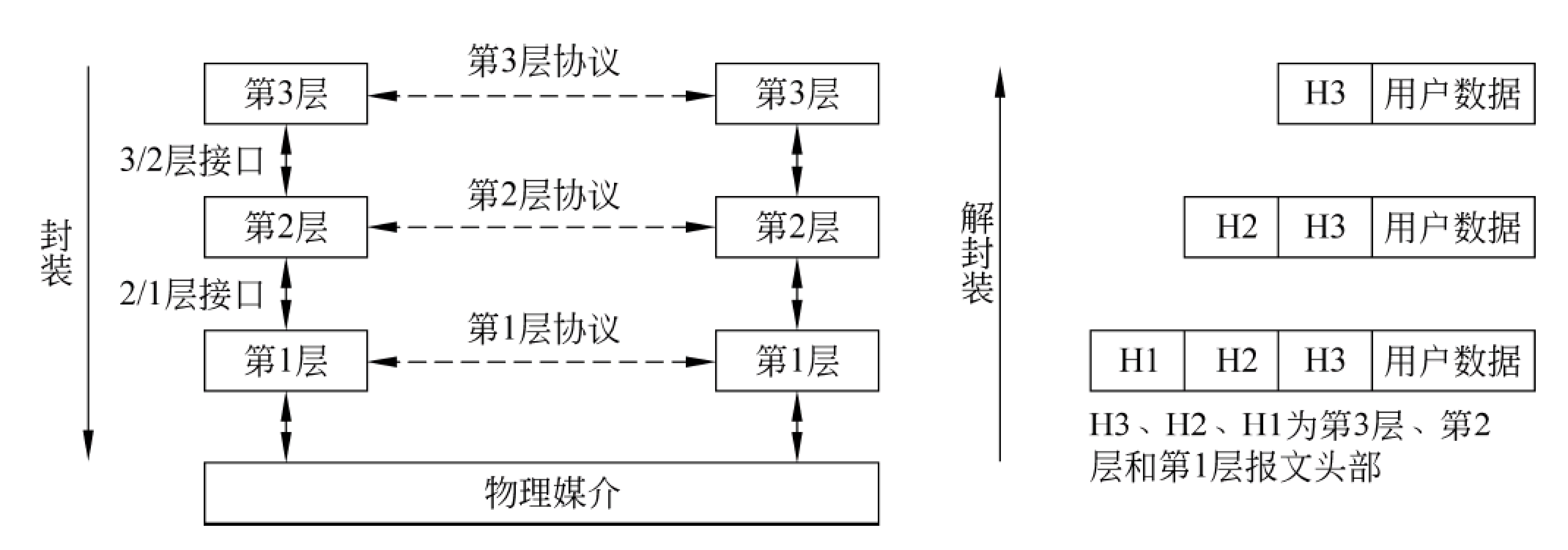


图 4-5 协议分层、封装和虚拟通信

在发送方，用户数据首先提交给第 3 层，第 3 层会在用户数据的前面添加本层的报文

头部 H3，然后作为一个整体（称为协议数据单元，Protocol Data Unit，PDU）提交给第 2 层；第 2 层又会在第 3 层的 PDU 前面加上报文头部 H2 作为第 2 层的 PDU 提交给第 1 层；同样，第 1 层会在第 2 层的 PDU 前面添加报文头部 H1，最后包含第 1、2、3 层报文头部和用户数据的报文被放到物理介质上传送给对方。一般地，网络中发送方的每一层都会在上层提交的 PDU 前面添加本层的报文头部后再提交给下层。

在接收方，从物理介质上接收到数据报文后，首先提交给第 1 层，在该层去掉第 1 层的报文头部，然后向上提交给第 2 层，在第 2 层又去掉第 2 层的报文头部，然后向上提交给第 3 层，在第 3 层又去掉第 3 层的报文头部，最后将用户数据提交给接收方用户（进程）。一般地，网络中接收方的每一层都会去掉 PDU 中本层的报文头部后再提交给上一层。

从这个例子可以看出，所谓对等层通信，并不是在通信双方的对等层之间直接传输数据，而是在发送方和接收方相邻的层次之间依次传递。因此，对等层之间的通信只是一种通过协议实现的虚拟通信。但正是在多层虚拟通信的支持下，数据才最终被传输到对方。而由于多层通信的存在，使得收发双方传输的数据报文中包含了多个报文头部，每个报文头部仅由与之相关的那一层进行处理。

在封装过程中，不同的协议，报文头部包含的信息也各不相同。一般来说要包含源地址、目的地址、数据长度、传输控制信息和校验信息等。

4. TCP/IP 协议及其体系结构

TCP/IP 是因特网所使用的标准协议。它实际上是一个协议簇，其中包含了很多具体的协议，它们共同构成了一个具有 4 个层次的体系结构，这 4 个层次从上到下分别是应用层、传输层、网际层和网络接口层，如图 4-6 所示。不过，其中的网络接口层并没有定义什么具体的内容，通常用 OSI/RM 中的数据链路层和物理层替代，所以也可以认为它具有 5 个层次。



图 4-6　TCP/IP 体系结构

应用层（application layer）为用户进程提供所需要的各种信息交换和远程操作服务。本层包含了很多面向应用的协议，如简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）、超文本传输协议（Hyper Text Transfer Protocol，HTTP）、文件传输协议（File Transfer Protocol，FTP）等。本层的 PDU 称为报文（message）。

传输层（transport layer）也称为运输层，为应用层进程提供端到端（不涉及中间结点）的通信服务。本层定义了两个主要的传输协议：无连接的用户数据报协议（User Datagram Protocol，UDP）和面向连接的传输控制协议（Transmission Control Protocol，TCP）。

UDP 在传送数据前不需要建立连接，如同邮递信件，属于发送后不管的方式，接收方接收到数据后也不需要给出任何确认，因此 UDP 提供的是一种不可靠的传输服务。但也正是由于不需要建立连接，因此协议开销小，灵活，资源占用少。UDP 协议的 PDU 称为用户数据报（user datagram）。

TCP 提供面向连接的服务。如同打电话，在传送数据前必须先建立连接，数据传送结束后要释放连接。面向连接意味着 TCP 协议提供的是可靠的传输服务，但也意味着需要增加许多任务开销，如确认、流量控制、计时器以及连接管理等。这不仅使报文头部增大很多，还要占用大量的处理机资源来处理这些任务。TCP 协议的 PDU 称为报文分段（segment）。

网际层（internet layer）也称为互联网层或网络层，为网络上的不同主机提供通信服务，即解决主机到主机的通信问题。网际层提供的是一种无连接、不可靠但尽力而为的传输服务。本层提供的核心协议是互联网协议 IP（Internet Protocol），还包括一些辅助协议，如地址解析协议（Address Resolution Protocol，ARP）、因特网控制报文协议（Internet Control Message Protocol）等。本层的 PDU 称为数据报，也常称为分组、封包或包（packet）。

网络接口层（network interface layer）提供了将数据报通过物理网络传输的服务。在概念上本层对应于 OSI/RM 的数据链路层和物理层，但是 TCP/IP 并没有规定本层的协议，其目的是使 TCP/IP 能够适应不同的物理网络。在实际应用中往往直接使用物理网络本身的相关协议。例如在局域网中主要采用 IEEE 802 系列协议，广域网常采用 HDLC、帧中继、PPP 等协议。

在 TCP/IP 网络中，物理网络是一个经常使用的概念。各种物理网络，如局域网、城域网、广域网甚至一条简单的物理线路，它们之间的差异可能非常大，但在 TCP/IP 看来，它们不过是主机之间的一条传输信息的“管道”而已。

4.1.3 网络应用模式

1. C/S 和 B/S 模式

C/S 模式即客户/服务器（Client/Server）模式，按这种工作模式构建的计算机网络相应地被称为 C/S 结构的网络。C/S 模式的基本思想是将网络应用分解成多个子任务，分别由客户端和服务器来完成。通常情况下，客户端完成数据表示和用户接口任务，服务器端完成业务处理和数据库访问任务。

在基于 C/S 模式的系统中，客户端接收用户的输入，然后向服务器端发出数据请求，服务器端根据请求查找数据库，取出数据进行处理，最后将结果传送到客户端，再由客户端对数据的表示形式和格式进行适当的转换呈现给用户。图 4-7 为 C/S 模式的示意图。

B/S 模式即浏览器/服务器（Browser/Server）模式。B/S 模式是随着因特网和万维网（World Wide Web，WWW）的发展而产生的。B/S 模式与 C/S 模式并没有本质的区别，B/S 模式可以认为是基于特定通信协议（HTTP）并采用网络浏览器作为客户端的 C/S 模式。它是为了满足瘦客户端（以网络浏览器作为客户端）的需要而产生的，可以大大节省客户端更新和维护的成本。但 B/S 模式仍存在着服务器负荷较重的问题。

2. P2P 模式

尽管 P2P（Peer-to-Peer）发展至今有较长一段时间，但学术界、工业界都没有形成一

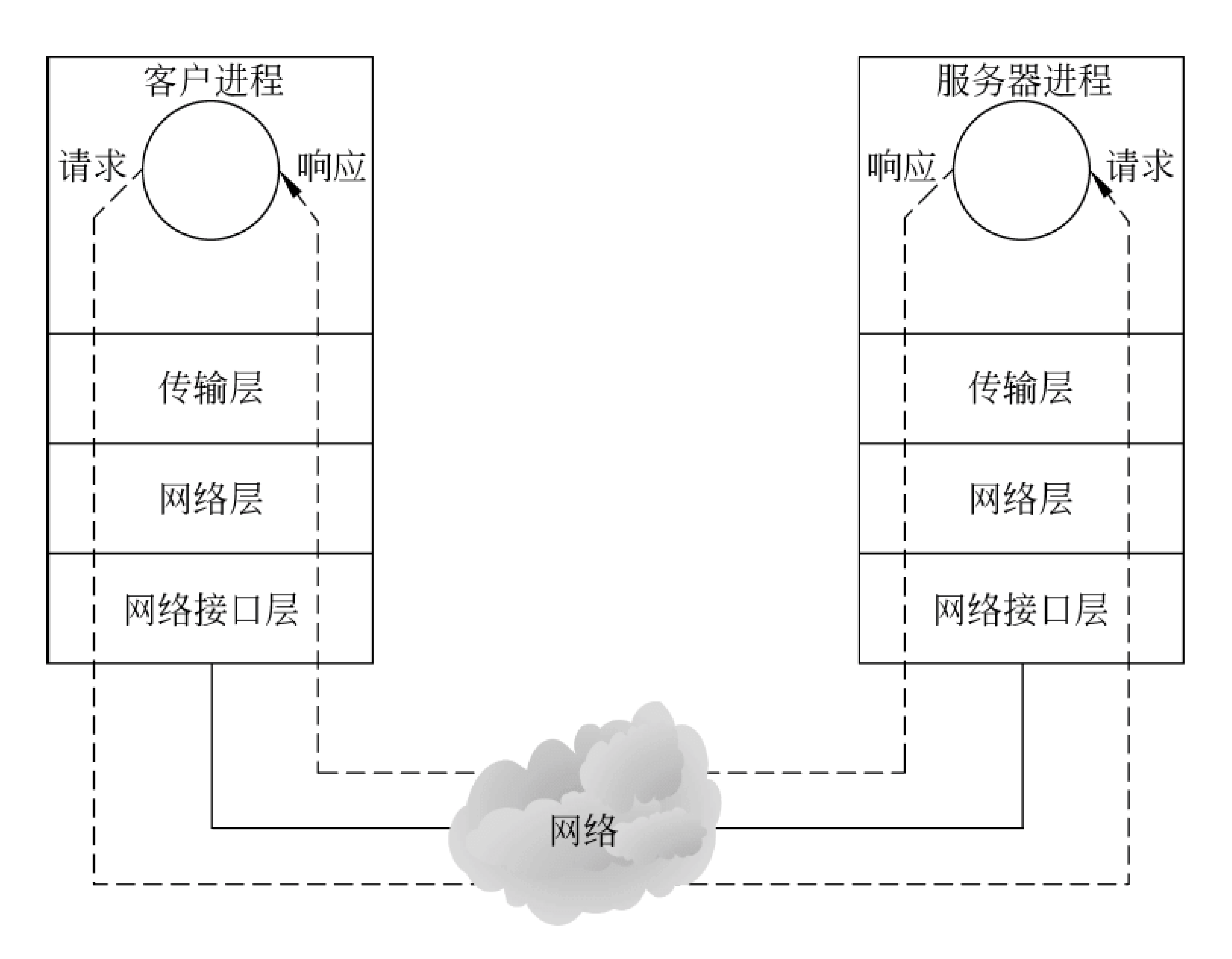


图 4-7　C/S 工作模式

个统一的完整定义。总体而言，现有的各种 P2P 定义要么是基于网络视图，要么是基于应用视图。前者认为 P2P 是一种用户之间通过某一相同的网络应用程序联系起来，彼此之间可以相互访问、共享计算机资源的网络；后者定义 P2P 是一种不通过中央服务器而将一些独立的计算机资源组织起来，通过 Internet 运行于个人计算机上，以实现共享文件和资源的应用。综合两种定义可以发现，P2P 本质上就是一种特殊网络，一种架构在 Internet 之上的应用层网络技术；其核心在于去除了中央服务器这一概念，将 Internet 建立在对等互联的基础上以实现最大程度的资源共享。

根据拓扑结构的关系可以将 P2P 网络分为 4 种形式：

- 集中式拓扑（centralized topology，也称作中心化拓扑）。
- 全分布式结构化拓扑（decentralized structured topology，也称作 DHT 网络）。
- 全分布式非结构化拓扑（decentralized unstructured topology）。
- 混合式拓扑（partially decentralized topology，也称作半中心化拓扑）。

上述 4 种 P2P 网络结构分别对应于 P2P 发展的不同时期，因而在其分类过程中，也可以按 P2P 发展的“代”来划分。集中式 P2P 网络常对应于第一代 P2P；而第二代 P2P 则是分布式网络体系结构，也叫纯 P2P 网络结构；第三代 P2P，就是混合式的 P2P 网络结构。在这种划分中，把当前正在发展的 P2P 技术归于第四代。

1）集中式 P2P 网络

在集中式 P2P 网络中，如图 4-8 所示，网络的主体由一个处于中心地位的目录服务器

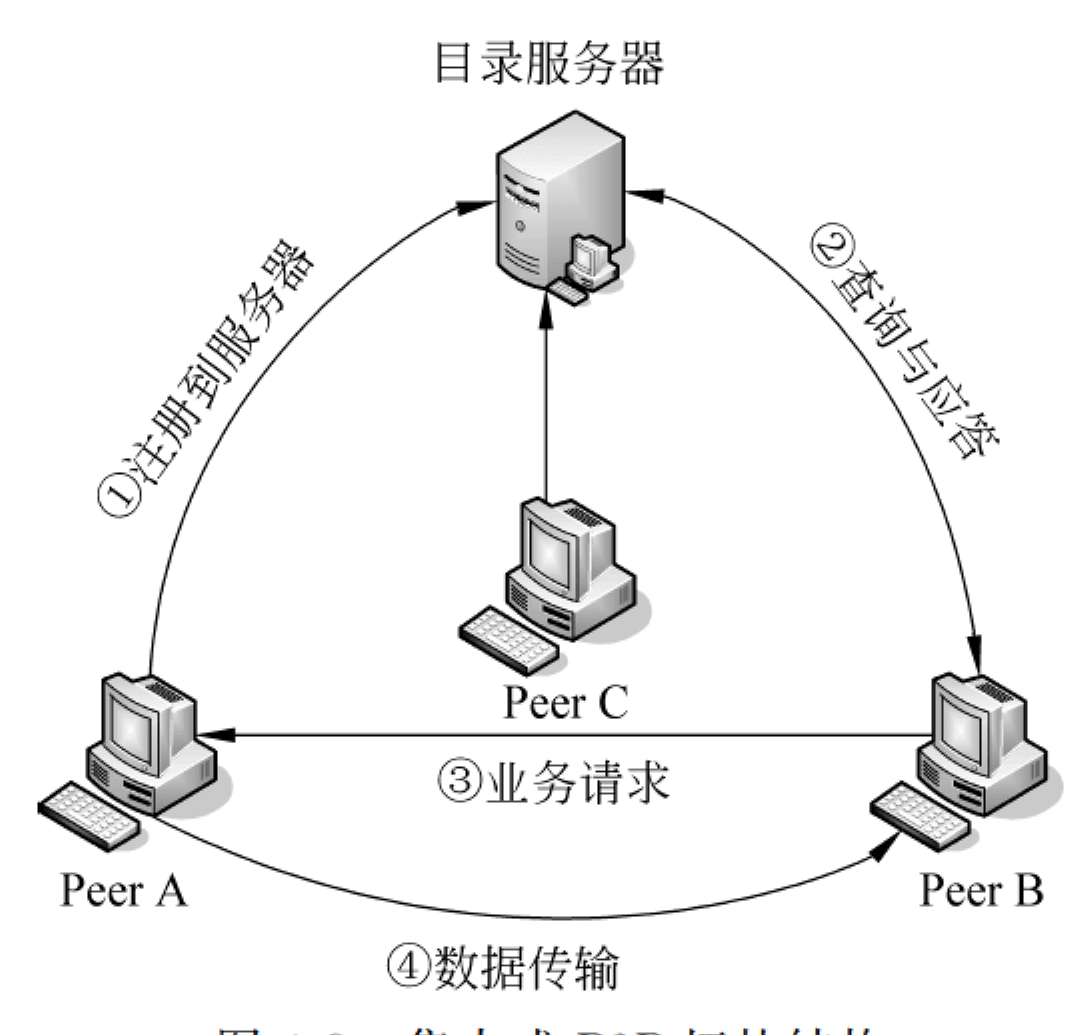


图 4-8　集中式 P2P 拓扑结构

(也称为索引服务器)和连接到目录服务器的各客户端结点(即 Peer)组成,目录服务器用来管理和组织 P2P 网络中的各客户端结点。在网络运行过程中,P2P 结点需要向中央目录服务器注册关于自身的信息,如名称、地址、资源和元数据等,但这些注册的所有内容都分散地存储在各个结点而非服务器上。查询时,结点根据目录服务器的应答以及网络流量和延迟等信息来选择和定位其他对等结点,并直接与之建立连接以传输资源,不必经过中央目录服务器进行。

集中式 P2P 网络最大的优点是维护简单,有效提高了网络的可管理性。在资源的搜索与发现方面,由于资源的发现依赖中心化的目录服务器,发现算法灵活、高效并能够实现复杂查询,这使得对共享资源的查找和更新非常方便,资源发现效率高。集中式 P2P 也存在不足,最大的问题与传统 C/S 工作方式类似,容易造成单点故障,主要表现在:中央服务器的故障容易导致整个网络的崩溃,可靠性和安全性较低;随着网络规模的扩大,当 Peer 结点大量增加时,系统的性能会大大下降;对目录服务器进行维护和更新的费用将急剧增加,所需成本过高;中央服务器的存在容易引起共享资源在版权问题上的纠纷。

2) 结构化 P2P 网络

全分布式结构化 P2P 网络主要是采用分布式散列表(Distributed Hash Table, DHT)技术来组织网络中的结点。DHT 是分布式系统中的一种,它将一个关键值(key)的集合分散到所有分布式系统结点上,能够有效地将信息转送到唯一拥有查询者请求关键值的结点上。

关键值分割是 DHT 中最基本的思想,大多数分布式散列表都使用某些稳定散列函数将关键值映射到结点中。稳定散列具有一个基本性质,即增加或移除结点只需改变邻近结点所拥有的关键值集合即可,其他结点则保持不变;而传统的散列表若要增加或移除一个结点,则需要重新映射整个关键值空间。

DHT 技术可以将广域范围内大量的结点共同形成并维护一个巨大的散列表,散列表被分割成不连续的块,每个结点被分配给一个属于自己的散列块,并成为这个散列块的管理者。结点利用 DHT 可以提供 P2P 网络中分布于众多结点中的总体视图,独立于实际位置。因此,数据的位置依赖于当前的 DHT 状态,而不是数据本身。

在 P2P 网络中,DHT 结构能够自适应结点的动态加入和退出,有着良好的可扩展性、鲁棒性、结点 ID 分配的均匀性和自组织能力。由于重叠网络采用了确定性拓扑结构, DHT 可以提供精确的资源发现,只要目的结点存在于网络中,DHT 总能找到它,资源发现的准确性得以保证。

3) 非结构化 P2P 网络

非结构化 P2P 网络一般采用基于完全随机图的组织方式,Gnutella 是全分布式非结构化 P2P 系统的典型实例,它是一个 P2P 文件共享系统。图 4-9 表示了 Gnutella 的网络结构模型中查询文件的洪泛流程,当一个客户端要下载一个文件时,需要经历如下几个步骤:

(1) 查询主机首先以文件名或者关键字生成一个查询。

(2) 查询主机将生成的查询发送给与它相邻的所有计算机。

(3) 这些直连的计算机如果有客户端请求的文件,则与查询主机建立直接连接;如果没有客户端请求的文件,则继续朝与自己相邻的计算机上转发这个查询。

(4) 重复以上过程,直到遍历完整个网络或找到客户端请求的文件为止。

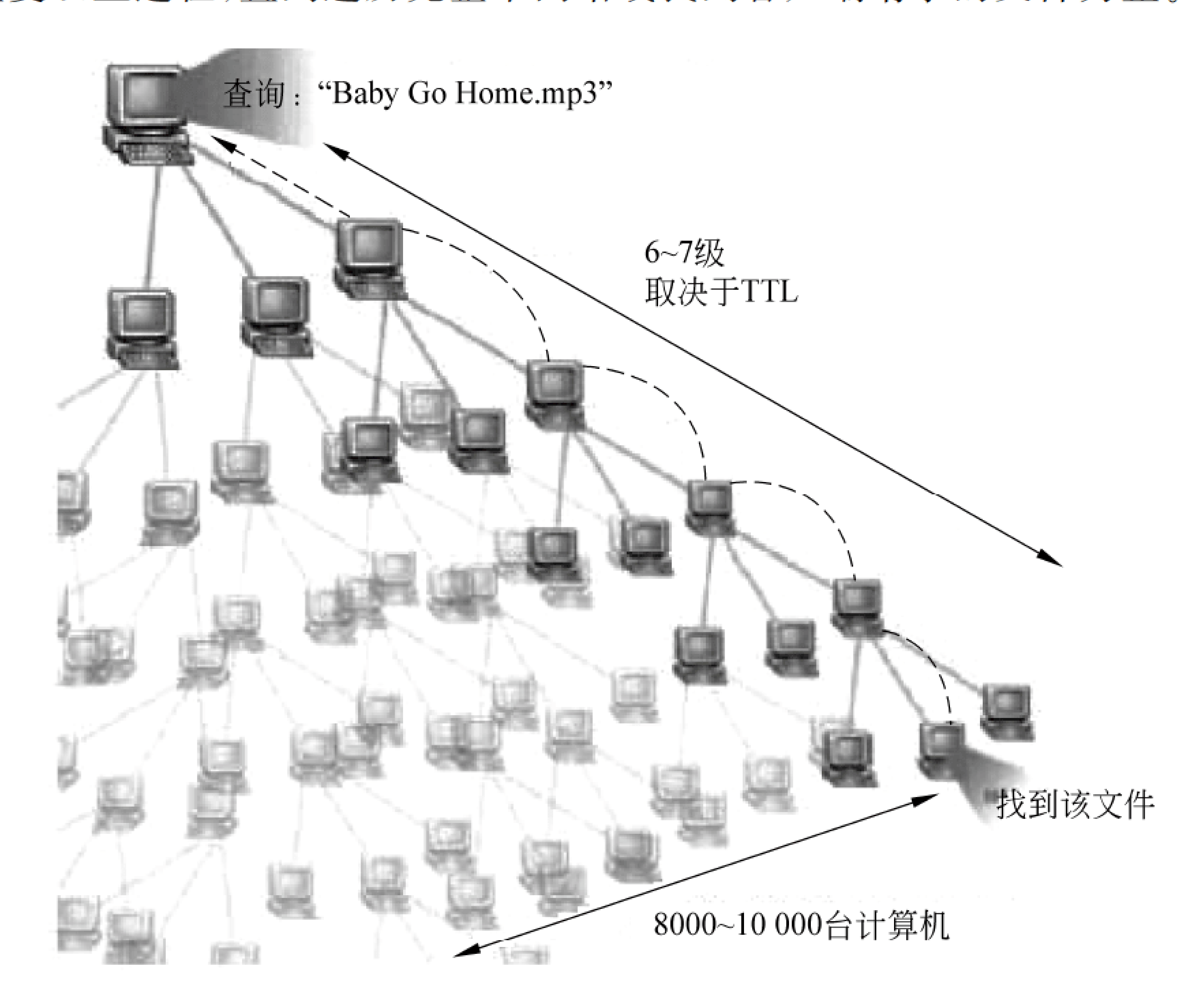


图 4-9　全分布式 Gnutella 网络结构

为了防止查询消息的传输范围过大,尤其是陷入循环传递困境,需要设置 TTL (Time To Live,生存时间)来控制查询的传递深度。不过,这有可能导致查询消息在传递到目标结点之前被丢弃。

非结构化 P2P 网络对信息定位没有严格要求,信息自由存储,构建简单,适合于信息发布、即时通信等结点随时加入和退出的应用场合。由于每一个结点都是一台普通的计算机,经常会连接或者断开网络,所以非结构化 P2P 网络结构不是完全稳定的,无法保证资源发现的效率。Gnutella 网络搜索的带宽消耗随着连接用户数目的增加而呈指数递增,经常饱和的连接会导致较慢的结点失去作用。因此,搜索请求在网络中会被经常丢弃,与整个网络相比,大多数的查询只会到达其中的很少一部分结点。

3. 应用模式比较

1) 网络结构

P2P 网络中每个结点的地位都是对等的,每个结点既充当服务器,为其他结点提供服务;同时也可以充当客户端享用其他结点提供的服务。P2P 模式与 C/S 模式的区别可以通过表 4-1 的对比得以说明。

从表中可以清楚地看出,P2P 模式与 C/S 模式是各有优劣的,不同的网络特性对应着不同的应用,在选择网络模式时应该根据应用的侧重点灵活选择合适的工作模式。

表 4-1　P2P 模式与 C/S 模式比较

比较要素	P2P 模式	C/S 模式	比较要素	P2P 模式	C/S 模式
数据分发	好	差	动态性	好	差
数据传输	一般	好	自组织性	好	差
数据交互	好	差	可扩展性	好	差
数据容量	好	差	抗干扰性	好	差
数据实时性	好	差	安全性	差	好
数据质量	一般	好	可管理性	差	好
数据覆盖率	差	好	成本控制	好	差
容错能力	好	差			

2）资源传输

在 Web 网络中，当用户间要进行信息资源的传输活动时，首先要构建一个有一定资源的站点，然后在其他 PC 上创建信息并“发布”到站点上。这些信息在站点上等待请求，接收到请求之后，站点将信息传递给请求者。整个过程需要 3 步，即：创建资源→发布资源→接收资源。同样是进行信息资源的传输，P2P 的处理过程则不同。处于对等地位的各个 PC 可以直接向存储着源文件的另一对等设备发送请求；而接收到请求的 PC 无须经过第三方中介，可以直接将需要的信息发送给对方，整个过程只需两步，即：创建资源→接收资源。

表 4-2 给出了 P2P 结点与 Web 站点在资源传输方面的比较，从中可以看出，P2P 改变了 Web 存在的信息消费者和生产者之间的不平衡，它允许更多的用户参与交换，且交换内容的过程和服务提供的方式都更加简便、直接；大量的信息不再集中于中心化的站点，而是被分布在多个对等设备中，这样，即使其中的一部分设备停止工作，其他参与设备仍然可以完成传输任务，这种理念正是 Internet 创建之初所追求的。

表 4-2　P2P 与 Web 的资源传输比较

比较要素	P2P	Web
资源交换对象间关系	对称	非对称
用户数量	越多越好	有上限值
内容和服务的提供	简便	复杂
网络可扩展能力	无限制	有容量限制
隐私保护能力	较弱	较强
资源数量	无上限	有上限
交互的工具	特定应用软件	通用浏览器
请求类型	大量不同的动态请求	大量相似请求
运行方式	对等方式	C/S
资源分配	分布式	集中式
传输协议	任意	基于 Web 传输协议
机器类型	任意	大量服务器

3）请求方式

在基于 C/S 模式的 Web 网络中，对服务、资源的请求基本上是一步完成的。当客户端向服务器请求一个文件时，直接通过统一资源定位符就可以将请求信息发送给服务器，服务器在接到请求后进行相应处理，并将处理结果直接返回给用户，这种请求方式是简单的一问一答，中间过程不会发生请求的转发或是目标的跳转。

在 P2P 网络中，对资源或服务的请求则需要多步才能完成。当一个结点需要特定资源时，它会向所有与之相连的其他结点发送同样的请求，如果被请求的结点中存在一个结点拥有被请求的资源，那么该结点就与请求结点直接建立连接并进行资源传输；如果这些直接连接的结点没有被请求的资源，则将此请求转发给与自己相连的其他结点，并依次扩散。请求信息会根据不同的搜索策略和算法规则在 P2P 网络中持续转发下去，只要 P2P 网络中存在被请求资源的结点，则请求最终总会被转发到目标结点上。

4）综合评价

C/S 模式造就了互联网的辉煌时代，然而 C/S 模式的结构特点和其内在的本质特性造成了互联网络上资源和服务的有向集中，无论信息资源还是成本资源均向同一方向集中。这种对资源的“中央集权”式的管理虽然与互联网的基本理念有些出入，但这种模式却符合一对多、强对弱的社会关系形式，如政府对个人和企业、大企业对小企业、学校对学生、企业对职工等。所以 C/S 模式也是符合市场需求的，不会因 P2P 的出现和兴起而走向没落，在可预见的将来仍将有更广、更深的发展。

作为近几年成长起来的 P2P 技术，不论是 P2P 的结构方式还是其表达的互联网思想与理念，都是最大限度地将信息数量、成本资源向互联网应用参与结点均匀分布，也就是所谓“边缘化”的趋势。此模式符合“一对一”的特点，也符合彼此相当的社会关系形式，如个人对个人、集团对集团、规模相当的企业之间等，这也是符合市场需求的。所以 P2P 与 Web 这两种方式不是你死我活的竞争关系，而是相互依赖、相互补充的共存关系，有关 P2P 即将替代 C/S 模式的说法是没有事实根据的。P2P 有其独特的市场空间，是对现有互联网应用的补充和增强。

4.2 因特网

因特网(Internet)是由成千上万不同类型、不同规模的计算机网络组成的世界范围的超大型计算机网络，使用 TCP/IP 协议作为其核心协议。

4.2.1 因特网基础知识

1. 因特网的结构与组成

因特网的结构按逻辑功能和工作方式划分为边缘部分和核心部分两个组成部分（见图 4-10）。

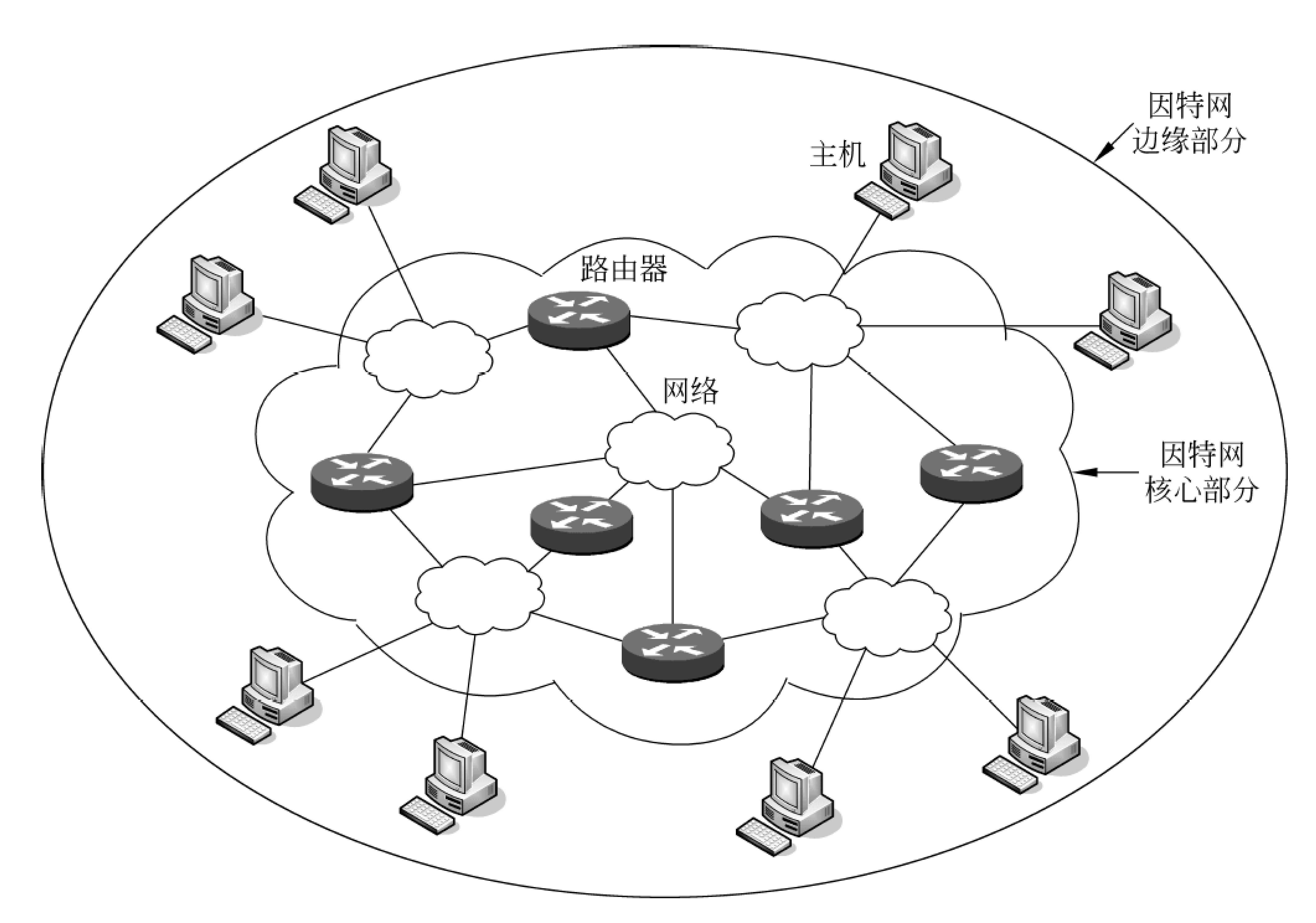


图 4-10　因特网的结构和组成

边缘部分由所有连接在因特网上的主机组成，是用户运行各种网络应用的终端设备。

核心部分由大量的网络和连接这些网络的路由器组成，是为边缘部分提供数据传输服务的设施。

1）因特网的边缘部分

连接在因特网上的所有主机构成了因特网的边缘部分。这些主机可以是普通的 PC、智能手机或平板电脑，还可以是非常昂贵的服务器或巨型计算机。主机的拥有者可以是个人，也可以是机构单位（学校、企业、政府机关等），当然也可以是某个 ISP（Internet Service Provider，因特网服务提供商）。边缘部分利用核心部分所提供的服务，使众多主机之间能够互相通信并交换或共享信息。为了区别因特网中的主机，因特网中的每个主机都有一个 IP 地址（更准确地说，主机中的每个网络接口都有一个 IP 地址）。

2）因特网的核心部分

因特网核心部分是因特网中最复杂的部分，核心部分要向因特网边缘中的大量主机提供连通性，使边缘部分中的任何一台主机都能够与其他主机进行通信。

因特网核心部分包含了大量的网络，这些网络可以是广域网，也可以是局域网，还可以是各种无线移动网络，所使用的数据通信技术也多种多样。所有的网络都通过一种称为路由器（router）的设备连接在一起，路由器能够将接收到的分组按最佳路径转发到另一个合适的网络。因为因特网是由大量相互连接的网络构成的，所以人们也将其称为互联网。为了识别因特网中的网络，因特网中的每个网络都有一个唯一的网络地址（格式与 IP 地址相同）。

从图 4-10 可以看出，一个网络的主机与另一个网络的主机之间的数据传输要跨越多个通过路由器连接的网络，而这之间的传输路径不只一条，因此需要路由器选择一条最佳

路径进行传输，选择最佳路径的策略和操作称为路由选择(routing)。

路由器的工作原理是，路由器中建有一个存放到达其他网络的表格——路由表(routing table)，当路由器从某个网络接收到一个分组时，首先取出该分组头部中的目的地址，然后根据目的地址查找路由表，找出应该从哪个网络将分组传送到下一个路由器，找到后就从连接该网络的接口转发该分组。路由表通常由路由器自动建立，建立路由表的算法称为路由算法(routing algorithm)，而实现路由算法的协议称为路由协议(routing protocol)。

2. 因特网接入

因特网核心部分所连接的网络可以是广域网，也可以是局域网。主机与因特网核心部分连接的接入技术根据主机的使用环境可以采用宽带接入、局域网接入或移动网络接入。

1）宽带接入

宽带接入采用的技术属于广域网技术，典型的宽带接入技术有 ADSL、HFC 和 FTTx。

ADSL(Asymmetric Digital Subscriber Line，非对称数字用户线)接入主要用于家庭用户，ADSL 是一种上、下行传输速率不相等的传输技术。上行传输是从用户到 ISP 方向的传输，下行传输是从 ISP 到用户方向的传输。ADSL 的下行速率要远远大于上行速率，故被称为非对称数字用户线。ADSL 的技术特征如下(参考图 4-11)：

- 使用普通电话线，无须另外架设线路。
- 能够在一条电话线上同时提供语音服务(打电话)和数据通信服务(上网)。
- 下行传输速率最高可达 8Mb/s，上行传输速率最高可达 1Mb/s，实际速率与用户到 ISP 之间的距离和电话线路质量有关。
- 用户端需安装用于上网的 ADSL 调制解调器和用于分离语音和数据的语音分离器。

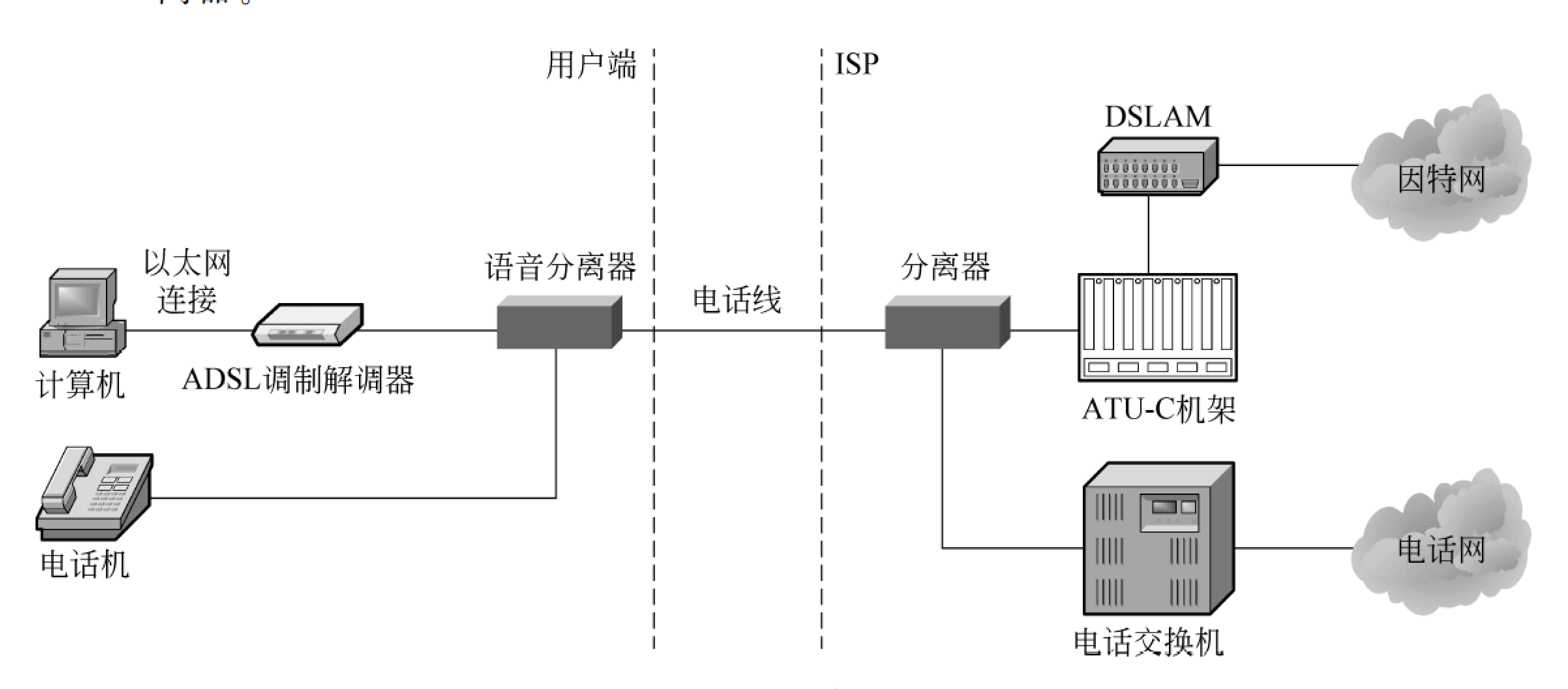


图 4-11　ADSL 接入配置图

HFC(Hybrid Fiber Coax)又称为光纤同轴混合网,它是在闭路电视网 CATV 的基础上开发的一种家庭宽带接入技术。除可传送电视节目外,HFC 网还提供电话、上网和其他宽带交互型业务。

HFC 的主要特点如下(参考图 4-12):

- 主干线路采用光纤,并使用模拟光纤技术进行传输。分支结点到家庭的线路仍然使用 CATV 同轴电缆。
- 下行传输速率最高可达 30Mb/s,上行传输速率最高可达 10Mb/s。
- 采用介质共享方式使用上行信道。
- 用户端需安装用户接口盒和电缆调制解调器(cable modem)。用户接口盒用于连接电视机的机顶盒、电话机和电缆调制解调器。电缆调制解调器是用户主机通过 HFC 网络接入因特网的设备。

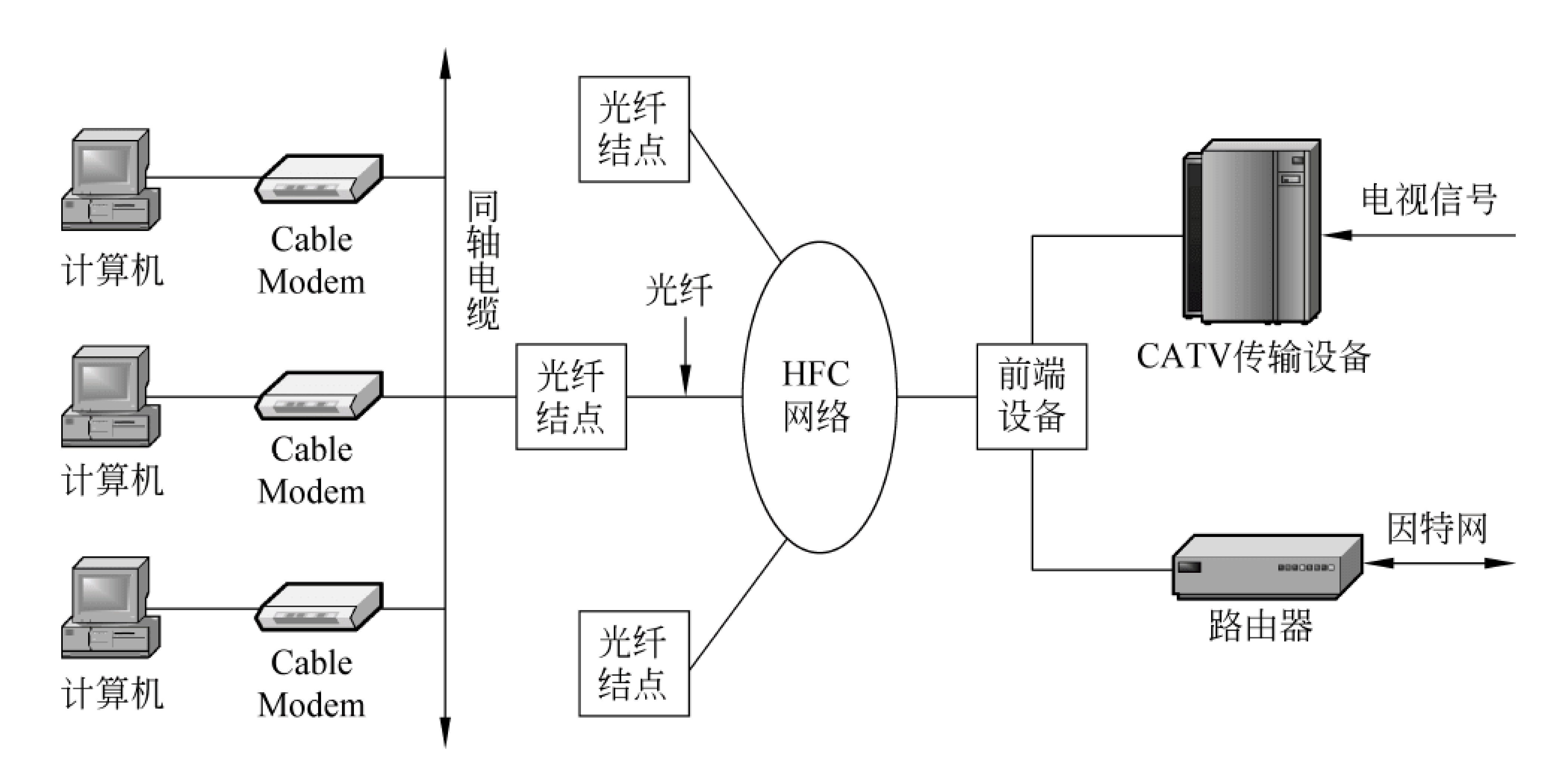


图 4-12　HFC 网接入配置图

FTTx 是一种使用光纤的宽带接入技术。FTTx 中的字母 x 代表不同的地点。FTTx 的传输速率可达 155Mb/s。

- FTTH(Fiber To The Home,光纤到户)即将光纤一直铺设到用户家庭,这可能是家庭用户接入因特网的最佳方法。但由于费用问题和使用效率问题,目前将光纤铺设到每个家庭还无法全面普及。
- FTTB(Fiber To The Building,光纤到楼)即将光纤铺设到楼宇,光纤进入楼宇后就转换为电信号,然后用双绞线分配到各用户。这种方案可支持大中型企业、商业或大公司高速率的宽带业务需求。
- FTTC(Fiber To The Curb,光纤到路边)实际上指的是光纤到居民区。光纤铺设到居民区后可为周边的一个或多个住宅小区的用户提供因特网接入服务。各个用户小区内部可使用局域网接入技术。

2) 局域网(以太网)接入

由于以太网技术并不适用于远程通信,所以严格意义上的以太网接入方式是不存在的。但如果不是很严格,则凡是用户通过单位或小区的以太网来访问因特网的方式也可以称为以太网接入。这时,以太网必须与其他宽带接入方式相结合。

3）移动网络接入

主要的移动网络接入技术包括 WiFi 接入和电信网络的 2G/3G/4G 接入技术。移动网络接入非常适合手机用户、平板电脑用户、便携式电脑用户或无法使用有线连接的用户。

2G/3G/4G 网络均采用蜂窝移动通信技术，2G 即第 2 代移动通信技术，包括 GSM（GRPS）和 CDMA 两种标准，数据传输率仅为 20～40kb/s。3G 即第 3 代移动通信技术，包括 W-CDMA、CDMA2000 和 TD-SCDMA 三个标准，数据传输速率可达 21Mb/s。4G 是最新的第 4 代移动通信技术，包括 LTE-TDD、LTE-FDD、WiMAX 等技术，数据传输速率理论上可达 100Mb/s。

WiFi 网络是符合 IEEE 802.11 标准的无线局域网（WLAN）。WiFi 网络的覆盖范围较小，所以仅在接入点（也称为热点，类似于手机通信中的基站）周围信号较好，连接速度较快。远离接入点时信号很差，可能会无法上网。

WiFi 接入的理论速度视接入点和用户终端采用的 IEEE 802.11 标准而定，IEEE 802.11b 的最大速率为 11Mb/s，IEEE 802.11g 的最大速率为 54Mb/s，IEEE 802.11n 的最大速率为 600Mb/s，最新的 IEEE 802.11ac 的最大速率为 1300Mb/s。

3. 因特网的数据传输过程

通过前面的叙述已知，因特网是一种采用 TCP/IP 协议的分组交换网，即用户数据被划分为小的分组在网上进行传送。下面以一份报文的传输为例介绍因特网中信息是如何传输的。注意，为了突出主线，此例中有些细节已进行了适当简化。

【例 4-3】 因特网中报文的传输过程。

本地主机要发送一份报文到远端主机时，将会执行以下操作：

（1）应用进程将要发送的报文提交给传输层。

（2）传输层向远端主机发起 TCP 连接，连接使用 4 个参数来描述：本地主机 IP 地址、本地应用进程端口、远程主机 IP 地址和远端应用进程端口。其中，端口标识了双方通信的进程是什么。

（3）TCP 连接建立起来后，TCP 把要传输的报文分解为多个报文分段，每个报文分段都加上 TCP 头部（其中包括本地进程端口和远端进程端口）形成 TCP-PDU，然后依次提交给网际层。

（4）在网际层，IP 协议将 TCP-PDU 加上 IP 头部（其中包括本地主机和远端主机的 IP 地址）形成 IP 分组（IP-PDU），然后将分组发送到网络上。

（5）因特网中的一系列路由器都会按照分组中的目的主机 IP 地址寻找最佳传输路径，并依次从一个网络传送到另一个网络，最终送达远端主机所在的网络，并被网络中的目的主机所接收（目的主机会看到分组中的目的 IP 地址是自己）。

（6）目的主机将收到的分组依次提交给网际层。

（7）网际层剥离掉分组前面的 IP 头部，将其中的 TCP-PDU 提交给传输层。

（8）传输层剥离掉 TCP-PDU 前面的 TCP 头部，将报文分段放入数据缓冲区。当所有的报文分段都正确接收后，传输层将它们组装成报文提交给应用进程（TCP 知道 TCP

头部中的端口号对应的是哪个应用进程)。

(9) 双方断开 TCP 连接。

可以看出,以上报文的传输过程类似于日常生活中邮件的邮递过程,邮件要经过一系列封装和解封装:物品→邮件→邮包→集装箱→邮包→邮件→物品。并且在其投递过程中需要通过若干邮局的转发才最终到达收件人。

4. IP 地址和端口号

就像每个人都有一个居住地址一样,在因特网中,每台主机的每个网络接口都分配有一个唯一的 IP 地址,它是因特网主机接口的"居住地址"。由于大多数情况下,每台主机只有一个网络接口,所以可以认为网络接口的 IP 地址就是主机的 IP 地址。IP 地址主要用于识别因特网中的不同主机,它在 TCP/IP 体系结构中的网际层进行识别和处理。

1) IPv4 地址

IPv4 地址用 32 位二进制编码表示。由于 32 位地址不容易记忆,所以,人们往往将 32 位的 IP 地址分为 4 个字节,每个字节间用圆点"."分隔,并以等效的十进制数表示。这种表示法称为"点分十进制表示法"。因为一个字节所能表示的最小十进制数为 0,最大十进制数为 255,所以 IP 地址的范围在 0.0.0.0 到 255.255.255.255 之间。

例如,IP 地址 10000001.00000001.00000001.00000010 可表示为 129.1.1.2。

由于因特网中既有主机,也有大量的网络。为了使 IP 地址既能标识一个网络,又能标识一台主机,一个 IP 地址又被分为网络地址(号)和主机地址(号)两个部分。这就使得 IP 地址成为一种具有层次性的地址(人们的居住地址也是有层次的,不过层次更多而已:省-市-区县-街道-小区-楼-室)。

IP 地址有 A、B、C、D、E 5 类,其中常用的是 A、B、C 3 类,这 3 类地址的格式如图 4-13 所示(地址高位的几位固定编码用于区分地址类型)。

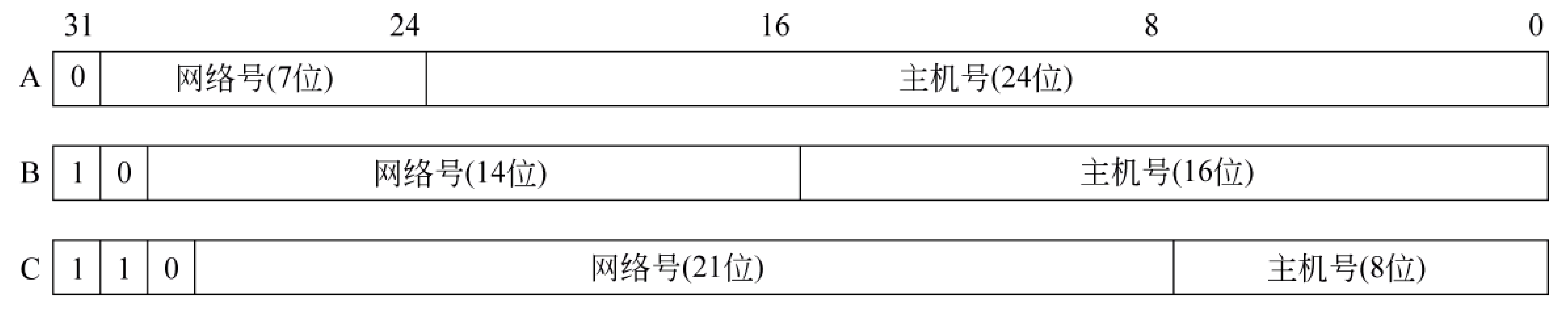


图 4-13　A 类、B 类、C 类 IP 地址的格式

A 类地址的网络号有 7 位,可提供使用的网络号有 126(2^7-2)个。减 2 的原因是,以下两个网络号不允许使用:

(1) 网络号为全 0 的 IP 地址是保留地址,意思是"本网络"。

(2) 网络号为 127(01111111)的 IP 地址保留作为本机软件环回测试(loopback test)之用。若主机发送一个网络号为 127 的 IP 分组,则自己将收到这个 IP 分组。

每一个拥有 A 类地址的网络(简称 A 类网络)允许的最大主机数是 16 777 214($2^{24}-2$),这里减 2 的原因是:主机地址全 0 表示"本主机",而全 1 表示"所有主机",即该网络上的所有主机。A 类地址主要分配给拥有大量主机的大型网络。

B 类地址的网络号有 14 位，但规定 128.0.0.0 不能使用，所以可供使用的网络号有 $2^{14}-1$(16 383)个。一个 B 类网络允许的最大主机数是 65 534($2^{16}-2$)，减 2 的原因同上。B 类地址主要分配给中等规模的网络。

C 类地址的网络号为 21 位，但规定 192.0.0.0 不能使用，所以可供使用的网络号有 $2^{21}-1$(2 097 151)个。一个 C 类网络允许的最大主机数是 254($2^{8}-2$)。C 类地址主要分配给规模较小的局域网。

表 4-3 给出了保留的特殊 IP 地址，其用途也在表中进行了说明。

表 4-3　保留的特殊 IP 地址

网络号	主　机　号	源地址	目的地址	含　　义
0	0	可以使用	不可使用	本网络上的本主机
0	host-id	可以使用	不可使用	本网络上的某主机(host-id 指定)
全 1	全 1	不可使用	可以使用	在本网络中进行广播
net-id	全 1	不可使用	可以使用	在 net-id 指定的网络上进行广播
127	全 0 或全 1 以外的任何数	可以使用	可以使用	本地环回地址

IP 地址中还有一些地址也不能在因特网中使用，这就是私有地址(private address)。私有地址是因特网规定专用于内网主机的 IP 地址。这里的内网是相对于公网(即因特网)而言的。内网是一些小型单位或家庭用户自己组建的局域网，其中的主机没有(或无法申请到)合法的 IP 地址，而私有地址就是专门为这类网络而保留的。私有地址不允许出现在因特网上，因特网中的路由器也不会转发那些具有私有地址的 IP 分组。

私有地址包括 3 个地址范围，这些地址都可以供内网自由选择使用：

10.0.0.0～10.255.255.255 (1 个 A 类地址块)

172.16.0.0～172.31.255.255(15 个 B 类地址块)

192.168.0.0～192.168.255.255(1 个 C 类地址块)

使用私有地址的主机需要通过一种称为网络地址转换(Network Address Translation，NAT)的机制才能访问因特网，NAT 可以将私有地址转化为因特网上的合法地址。NAT 通常设置在内网的网关中。NAT 是解决 IPv4 地址不足问题的临时性解决方案，它不适用于新的 IPv6 网络。

2) IPv6 地址

随着网络应用和管理需求的发展，IP 协议在地址空间、性能、安全性和自动配置方面的不足表现得越发明显。IP 地址危机由来已久，地址空间的局限性是促使 IP 协议升级的主要动力；尽管 IPv4 表现得不错，但一些源自 20 世纪 80 年代甚至更早期的设计有待进一步改进以提高网络性能；安全性一直被认为是由网络层以上的层次所负责，但现在已经成为 IP 的下一个版本可以发挥作用的地方；IPv4 结点的配置一直比较复杂，网络管理员与用户喜欢“即插即用”的配置方式，即：将计算机连接在网络上后就可以自动完成相关配置，尤其是 IP 主机移动性的增强进一步要求当主机在不同网络间切换或使用不同的网络接入点时能够提供更好的配置支持。

与 IPv4 相比较，IPv6 除了能够提供海量地址空间外，在技术上还有很多新的特点，

主要表现在如下几个方面。

IPv6 地址长度为 128 位，可以方便地进行网络的层次化部署。同一组织机构在其网络中可以只使用一个前缀；对于 ISP 而言，可以获得巨大的地址空间，这样 ISP 可以把所有客户聚合形成一个前缀并发布出去。分层聚合使得全局路由表的长度大大减小，提高了分组转发效率。另外，由于地址空间巨大，同一客户使用多个 ISP 接入时可以同时使用不同的前缀，这样也不会对全局路由表的聚合造成影响。

基于种种原因，IPv6 废弃了 IPv4 头部中的诸多控制域，IPv6 基本报文头的处理比 IPv4 大大简化，提高了处理效率。另外，IPv6 为了更好地支持各种选项处理，提出了扩展头的概念，新增选项时不必修改现有报文结构，理论上可以无限扩展，体现了协议的灵活性。

IPv6 内置支持通过地址自动配置方式使主机自动发现网络并获取 IPv6 地址，大大提高了内部网络的可管理性。自动配置功能使得用户设备，尤其是移动电话、无线结点等人机交互功能弱小的设备，无需手工配置或使用专用服务器（如 DHCP Server）。

尽管 IPv4 通过 IPSec 也能够支持 IP 层的安全特性，但只是通过选项支持，且实际部署中多数结点都不支持；而 IPSec 是 IPv6 基本定义中的一部分，任何部署的结点都必须支持。因此，在 IPv6 中支持端到端安全要容易得多。

IPv6 规定必须支持移动特性，任何 IPv6 结点都可以使用移动 IP 功能。和移动 IPv4 相比，移动 IPv6 使用邻居发现功能可直接发现外地网络并得到转交地址，而不必使用外地代理。同时，利用路由扩展头和目的地址扩展头，移动结点和对等结点之间可以直接通信，解决了移动 IPv4 的三角路由、源地址过滤等问题，使得移动通信效率变得更高。

与 IPv4 相比，IPv6 的地址比特数是 IPv4 的 4 倍，从原来的 32 位扩充到 128 位，128 位地址空间可包含约 43 亿×43 亿×43 亿×43 亿个地址，足以满足任何可预计的地址空间分配。为了使地址的表示简洁明了，IPv6 使用冒号十六进制表示法，它把每个 16 位的二进制值用十六进制值表示，各值之间用冒号分隔。例如，如果将前面所给的点分十进制数表示法的地址值改为冒号十六进制表示法，就变成了如下结果：

68E6:8C64:FFFF:FFFF:0:1180:960A:FFFF

在冒号十六进制表示法中，允许把数字前面的重复 0 省略，在刚才的例子中，就把 0000 中的前 3 个 0 给省略了。另外，冒号十六进制表示法还使用了两个技术使它的表示更为有效，首先，它允许零压缩，即一连串连续的零可以用一对冒号所取代，例如：

FF05:0:0:0:0:0:0:B3

可以简写成

FF05::B3

为了保证零压缩的一致解释，规定在任意一个地址中只能使用一次零压缩。该技术对已建议的地址分配策略特别有用，因为其中的很多地址都包含较长的连续零串。

其次，冒号十六进制表示法可结合点分十进制表示法的后缀使用，该技术在 IPv4 向 IPv6 过渡阶段非常有用，例如：

0:0:0:0:0:0:128.10.2.2

在上面地址表示方法中，尽管冒号分隔的每个值是两个字节的量，但每个点分十进制的值还是指明一个字节的值。再结合零压缩，即可得出如下的地址表示：

::128.10.2.2

为了方便地表示 IPv6 地址的网络前缀,CIDR 的斜线表示法仍然可以使用。网络前缀表示为 ipv6-address/prefix-length。其中,ipv6-address 为十六进制表示的 128 比特地址,prefix-length 为十进制表示的地址前缀长度。例如,60 位的网络前缀 12AB00000000CD3 可表示为

12AB:0000:0000:CD30:0000:0000:0000:0000/60

12AB::CD30:0:0:0:0/60

12AB:0:0:CD30::/60

但不能表示成如下形式:

12AB:0:0:CD3/60　　非法表示

12AB::CD30/60　　表示地址 12AB:0:0:0:0:0:0:CD30 的前 60 位网络前缀

12AB::CD3/60　　表示地址 12AB:0:0:0:0:0:0:0CD3 的前 60 位网络前缀

3) 端口号

用 IP 地址可以标识一台主机,但如果一台主机中同时有多个进程都要进行数据传输,那么这些不同的进程应该如何识别呢? 或者说,主机接收到的数据应该由哪个进程来处理呢? 标识同一主机中不同进程的机制是端口号(port number)。如果把 IP 地址看成是港口的地址,那么端口号就是该港口某个泊位的编号。

端口是传输层与应用层之间数据交换的一种机制。应用层进程在其运行期间总是与某个端口绑定,不同应用层进程的端口各不相同。传输层要将数据提交给某个应用层进程,只要知道该应用层进程对应的端口号即可。

端口在逻辑上可以看成是一个带有编号的数据"管道",一旦传输层与应用层进程之间通过某个"管道"实现了对接,二者之间就可以通过该"管道"实现数据交换。这与装运不同货物的货船(提供运输服务)进入港口后要停泊在指定编号的泊位(此泊位已与这个"应用"业务绑定)进行货物装卸的道理是一样的。例如,装运"FTP 数据"的货船要停泊在 21 号泊位(端口)装卸货物,装运"Web 数据"的货船要停泊在 80 号泊位装卸货物,装运"E-mail 数据"的货船要停泊在 25 号泊位装卸货物等。

网络中的端口号是一个 16 位的正整数(0～65 535),其中 0～1023 被固定分配给一些应用层协议(永远不会改变),这些端口号称为熟知端口号(well-known port number);1024～49 152 称为已注册端口号,被一些公司用于自己的某种协议;49 152～65 535 为临时端口号,留给应用进程临时使用。临时端口号使用后会被系统回收,可以再分配给其他应用进程重复使用。

一些常用的 TCP 熟知端口号所对应的应用层协议如下:

21　FTP(文件传输协议)

23　Telnet(远程登录)

25　SMTP(邮件传输协议)

80　HTTP(超文本传输协议)

110　POP3(邮局协议)

5. 子网和子网掩码

32 位的 IP 地址所能表示的网络数是有限的。随着因特网中网络数量的增长，网络地址不够的问题日益严重。IPv4 的解决办法是采用子网寻址技术，将主机地址空间划出一定的位数作为网络号的一部分，划分出来的那几位就称为子网号，而剩余的主机地址空间作为子网的主机地址空间，这样一个网络就被分成多个子网，使地址空间得到更有效的使用。

进行子网划分后，IP 地址就划分为“网络-子网-主机”3 部分。

【例 4-4】 将 C 类网络 202.117.58.0 划分成 4 个大小相等的子网。

202.117.58.0 的二进制编码为 11001010.01110101.00111010.00000000，其中高 24 位为网络号，最低 8 位为主机号。划分 4 个子网需要从主机号部分拿出 2 位用作子网号，因此 4 个子网的网络地址和主机地址范围如下。

子网 1：11001010.01110101.00111010.**00**000000(202.117.58.0)

子网 1 主机地址范围：202.117.58.1～202.117.58.63

子网 2：11001010.01110101.00111010.**01**000000(202.117.58.64)

子网 2 主机地址范围：202.117.58.65～202.117.58.127

子网 3：11001010.01110101.00111010.**10**000000(202.117.58.128)

子网 3 主机地址范围：202.117.58.129～202.117.58.191

子网 4：11001010.01110101.00111010.**11**000000(202.117.58.192)

子网 4 主机地址范围：202.117.58.193～202.117.58.254

划分子网后，网络是如何识别各个子网的呢？区分一台主机属于哪个子网，可以通过子网掩码(subnet mask)来实现。与 IP 地址相似，子网掩码也是一个 32 位的二进制数，也用点分十进制数表示。一个子网的子网掩码定义为：对应于 IP 地址中的网络号和子网号部分，子网掩码中相应的位为 1；对应于主机号部分，子网掩码中相应的位为 0。

【例 4-5】 写出例 4-4 中子网的子网掩码。

按照子网掩码的定义，例 4-4 中子网的子网掩码是

11111111.11111111.11111111.11000000(255.255.255.192)

对标准的 A、B、C 类网络，根据子网掩码的定义，可以知道它们的默认子网掩码如下：

A 类网络的默认子网掩码为 255.0.0.0。

B 类网络的默认子网掩码为 255. 255.0.0。

C 类网络的默认子网掩码为 255.255.255.0。

在计算机网络中，子网掩码是一个非常重要的参数，它涉及路由器接收到一个 IP 分组后如何转发该分组：

- 在源端网络，是否将该分组转发到因特网？
- 在因特网中，应从通过哪条路径转发该分组？
- 到达目的网络时，该分组应转发到哪个子网？

那么路由器是如何计算一个 IP 地址中的子网地址呢？方法很简单，只要将子网掩码与 IP 地址进行“与”运算，结果就是子网地址。根据子网地址的概念还可得出这样的结

论：如果两个主机的IP地址经过子网掩码运算后结果相同，则表示两台主机处于同一子网中。

【例4-6】 判断IP地址的子网属性：

（1）判断C类地址202.117.35.239和202.117.58.114是否属于同一子网。

（2）判断202.117.35.239和202.117.35.200是否属于同一子网（即网络号是否相同），已知子网掩码为255.255.255.192。

解：

（1）将十进制表示的两个C类IP地址转换为二进制表示。

202.117.35.239的二进制表示：11001010.01110101.00100011.11101111

202.117.58.114的二进制表示：11001010.01110101.00111010.01110010

C类IP地址的默认子网掩码为255.255.255.0，即

11111111.11111111.11111111.00000000

再分别用该子网掩码和两个IP地址进行逻辑“与”运算，运算结果如下

202.117.35.239的子网号为202.117.35.0。

202.117.58.114的子网号为202.117.58.0。

两者的子网号不同，因此这两个IP地址不属于同一子网。

（2）将IP地址和子网掩码写成二进制。

202.117.35.239的二进制表示：11001010.01110101.00100011.11101111

202.117.35.193的二进制表示：11001010.01110101.00100011.11000001

子网掩码255.255.255.192的二进制表示：11111111.11111111.11111111.11000000

再用子网掩码分别与两个IP地址进行逻辑“与”运算，运算结果如下：

202.117.35.239的子网号为202.117.35.192。

202.117.35.193的子网号为202.117.35.192。

两者的子网号相同，因此这两个IP地址属于同一子网。

6. 域名地址和MAC地址

在因特网中，除了用于主机寻址的IP地址和用于应用进程寻址的端口号，还有两类地址也是非常重要的：域名地址和MAC地址。

1）域名地址（简称域名）

尽管用十进制点分表示的IP地址比二进制形式的IP地址识别起来容易一些，但仍然不便于记忆，而且从表示形式上看不出拥有该IP地址的主机的用途。从1985年起，因特网在IP地址的基础上开始向用户提供域名到IP地址解析的域名服务（Domain Name Service，DNS），以方便用户使用便于记忆和识别的域名来标识接入因特网的主机。域名有固定的格式，并通过域名服务与IP地址进行了绑定。例如，西安交通大学Web服务器的域名是www.xjtu.edu.cn，该域名对应的IP地址是202.117.0.13。

DNS系统由3个部分组成：域名空间、域名服务器和解析程序。DNS将整个因特网视为一个域名空间，由不同层次的域（domain）组成。每个域都有自己独立的域名服务器，在域名服务器中存放着主机的域名与IP地址之间的对应关系表（域名数据库）。因

特网中有许多域名服务器，分布在世界不同的地方，它们之间通过特定的协议进行相互通信和联系，这就保证了用户可以通过本地的域名服务器查找到因特网上的所有域名信息。

域名由若干个分量组成，各分量之间用点“.”隔开。例如，一个具有3个层级的域名可以表示如下：

三级域名.二级域名.顶级域名

域名中的各分量分别代表不同层级的域名，层级最低的域名写在最左边(通常为主机名)，层级最高的顶级域名则写在最右边。例如，mail.xjtu.edu.cn表示西安交通大学的电子邮件服务器，其中，mail为邮件服务器主机名，xjtu为西安交通大学的域名，edu是教育科研领域的域名，最右边的顶级域名cn为中国的国家域名。要注意的是，域名只是逻辑概念，并不反映主机所在的物理地点。

图4-14是Internet域名空间的树形结构，最上面的树根没有名字。根下面是顶级域结点，再下面是二级域结点，依此类推，最下面的叶子结点是主机名。一些常用的顶级域名的含义如表4-4所示。

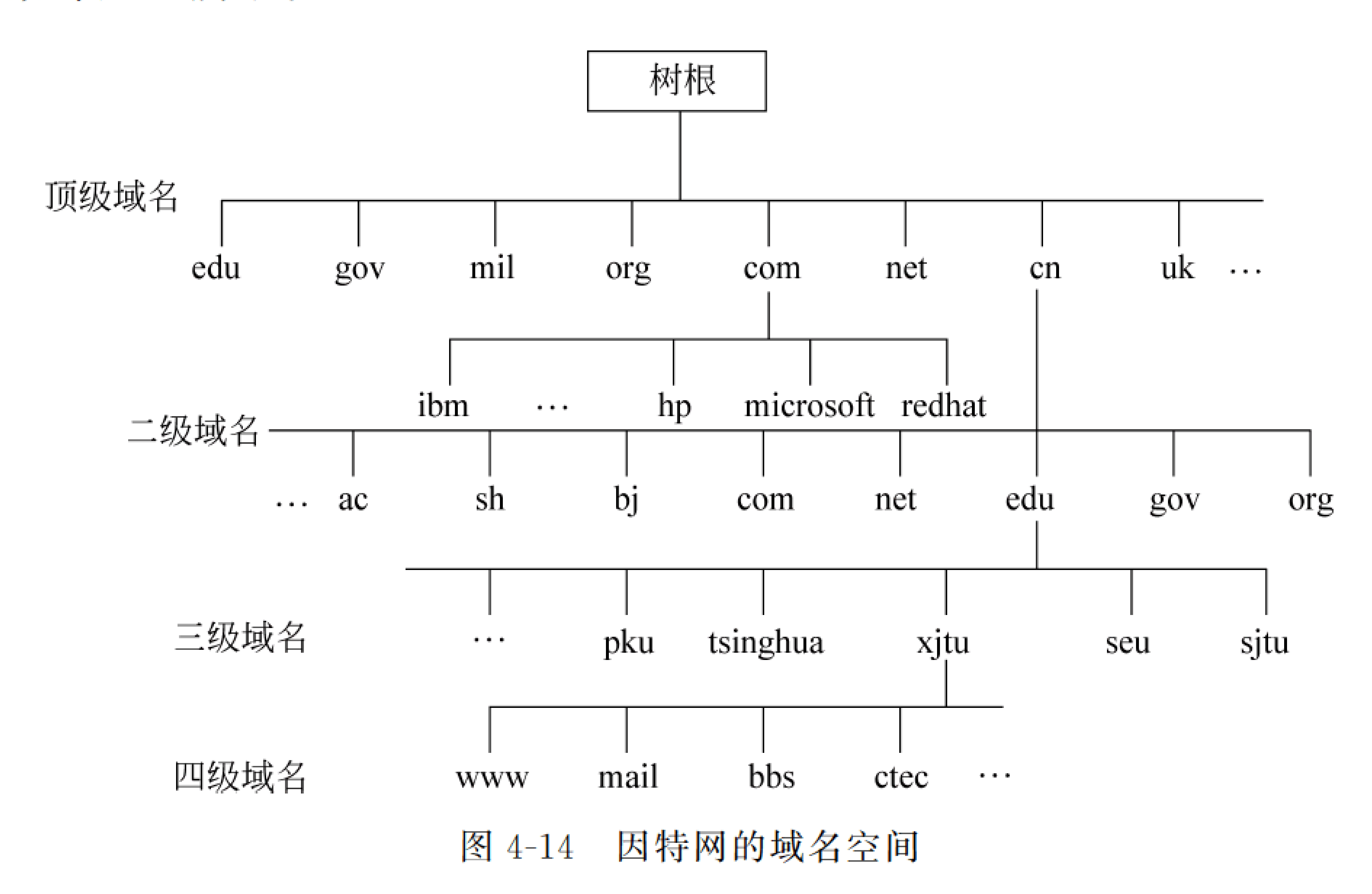


图4-14 因特网的域名空间

表4-4 常用顶级域名及其含义

顶级域名	含义	顶级域名	含义
.edu	教育机构	.com	商业组织
.gov	政府部门	.net	网络服务机构
.mil	军事部门	.cn	中国
.org	非营利性组织		

通过域名查找IP地址的过程称为域名解析(domain name resolution)。域名解析由相关的域名服务器共同完成，基本过程如下：

当某个主机的应用进程通过域名访问目的主机时，必须先将目的主机的域名转换为对应的IP地址。应用进程首先将待转换的域名放在DNS请求报文中，发给本地域名服

务器。本地域名服务器在本机的域名数据库中查找,如果没有找到,则将该 DNS 请求报文转发给顶级域名服务器,顶级域名服务器根据待查找的域名把 DNS 请求转发给相应的二级域名服务器,二级域名服务器再根据域名把 DNS 请求转发给相应的三级域名服务器,如此重复,当 DNS 请求送达负责管理该域名的域名服务器时,该域名服务器在域名数据库中找到请求的域名对应的 IP 地址,然后将 IP 地址封装在 DNS 应答报文中,逐级回送,最终送达发出 DNS 请求的应用进程,然后应用进程就可以用 IP 地址和目的主机进行通信。

在实际应用中,为了减少域名查找的开销,减轻上层域名服务器的压力,每个域的域名服务器(包括本地主机)中都会设置一个 DNS 缓存,将相关的域名查询记录保存一段时间,这样,对于同样的查询只要在 DNS 缓存中查找即可。

如果想知道域名所对应的 IP 地址,在 Windows 系统下,打开命令提示符窗口,输入 nslookup＜域名＞,就可以看到该域名对应的 IP 地址了。例如,在命令提示符窗口输入以下命令并按回车:

nslookup www.baidu.com

窗口中就会显示百度网站的域名 www.baidu.com 对应的 IP 地址,如图 4-15 所示。

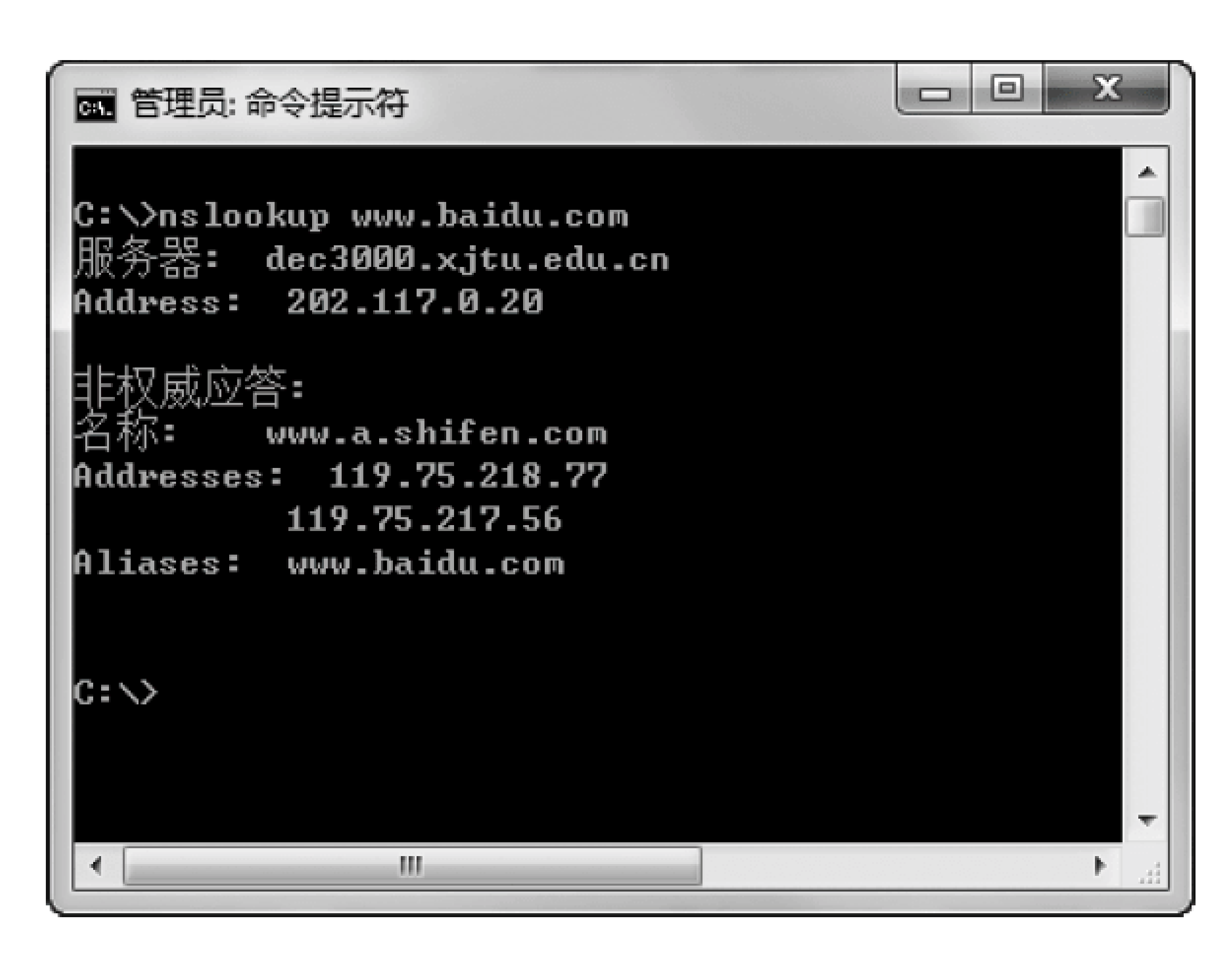


图 4-15　Windows 系统中的 DNS 查询命令

2) MAC 地址

MAC 地址是固化在网络接口硬件中的地址,也称为物理地址或硬件地址,用于标识一个主机的网络接口。MAC 地址在网络接口层(更确切地说是局域网数据链路层中的 MAC 子层)中使用。在局域网中,不同主机数据链路层之间的"虚拟"通信必须指定源主机和目的主机的 MAC 地址。

MAC 地址是一个 48 位的二进制编码(但以十六进制形式表示),如 00-15-58-EB-C1-DA。其高 24 位是设备的生产厂商编码,低 24 位是生产厂商内部的产品序列号。

MAC 地址和 IP 地址之间并没有必然的联系。MAC 地址就像一个人的身份证号,无论居住地在哪,身份证号永不会改变;IP 地址则如同一个人的居住地址(或邮政编码),搬家以后,其居住地址(或邮政编码)就随之发生改变。如果主机更换了一个新的网络接口,MAC 地址也随之改变,但主机的 IP 地址不会改变。反之,如果主机从一个网络移到

另一个网络（典型例子就是用笔记本电脑在家上网和在单位上网），其 IP 地址也要发生变化，但其 MAC 地址不会变化（因为网络接口没有更换）。

如果想知道本机的 MAC 地址，在 Windows 系统下，打开命令提示符窗口，输入 ipconfig /all，就可以看到本机的 MAC 地址（如图 4-16 所示）。注意，此命令也会显示本地主机的 IP 配置信息。

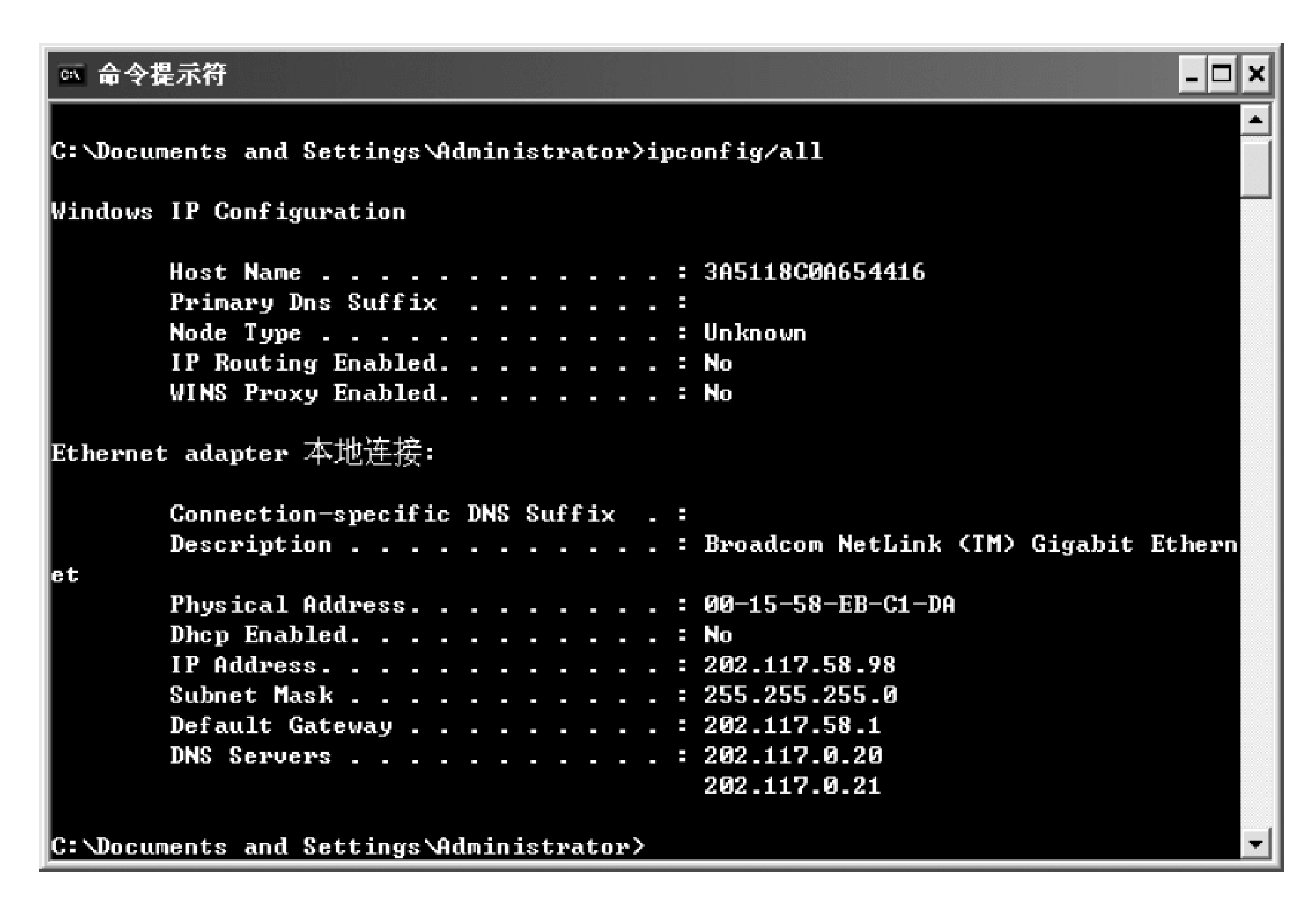


图 4-16　使用 Windows 的 ipconfig 命令查看本机的网络配置

4.2.2　常见的因特网应用

本节介绍计算机网络的通信服务是如何提供给应用进程来使用的，换句话说，各种应用进程通过什么样的应用层协议来使用网络所提供的通信服务。

应用层的许多协议都是基于 C/S 方式。这里再强调一下，客户和服务器都是指通信中所涉及的两个应用进程。C/S 方式所描述的是进程之间服务和被服务的关系。这里最主要的特征就是：客户是服务请求方，服务器是服务提供方。

本节讨论的内容包括电子邮件、万维网和文件传输。

1. 电子邮件

电子邮件（E-mail）是因特网上最基本、最常用的一种应用，它不仅可以传送邮件本身，还可以以附件形式传送文字、声音、图像、数值数据等内容。

要使用电子邮件，用户首先要拥有一个电子邮箱，用户使用电子邮件地址来访问自己的电子邮箱。电子邮件地址的格式为：用户名@用户的邮件服务器域名，例如 wgchen@stu.xjtu.edu.cn。

一个电子邮件系统主要由用户代理、邮件传送协议和邮件服务器 3 个部分组成。

用户代理（user agent）是用户和电子邮件系统的接口，为用户提供一个友好的收发邮件的界面。用户代理软件有很多，如 Windows 下的 Outlook、Outlook Express、Foxmail

等,Linux 下的 thunderbird 和 mutt 等。

邮件服务器提供了邮箱存储空间和邮件传送功能。邮件传送需要使用两种不同的协议:

(1) SMTP(简单邮件传输协议)用于用户代理向邮件服务器发送邮件和在邮件服务器之间传送邮件,它是电子邮件系统中邮件传输的标准应用协议,借助于传输层的 TCP 协议进行信息传输。

(2) 邮局协议 POP3(Post Office Protocol version 3)是用户代理从邮件服务器读取邮件的协议。在电子邮件系统中,用于存储和投递电子邮件的主机被称为 POP3 服务器。POP3 服务器与邮件服务器通常位于同一台主机。

邮件服务器是电子邮件系统的核心构件,其功能是发送和接收邮件,同时还要向发信人报告邮件传送的情况。邮件服务器按照 C/S 方式工作,它兼有客户和服务器两种角色,当它向目的邮件服务器发送邮件时,它就是 SMTP 客户;当它从用户代理接收邮件或从源邮件服务器接收邮件时,它就是 SMTP 服务器。因此,邮件服务器内部始终运行着两个进程,一个是 SMTP 客户进程,另一个是 SMTP 服务器进程。

电子邮件发送和接收的过程如图 4-17 所示。

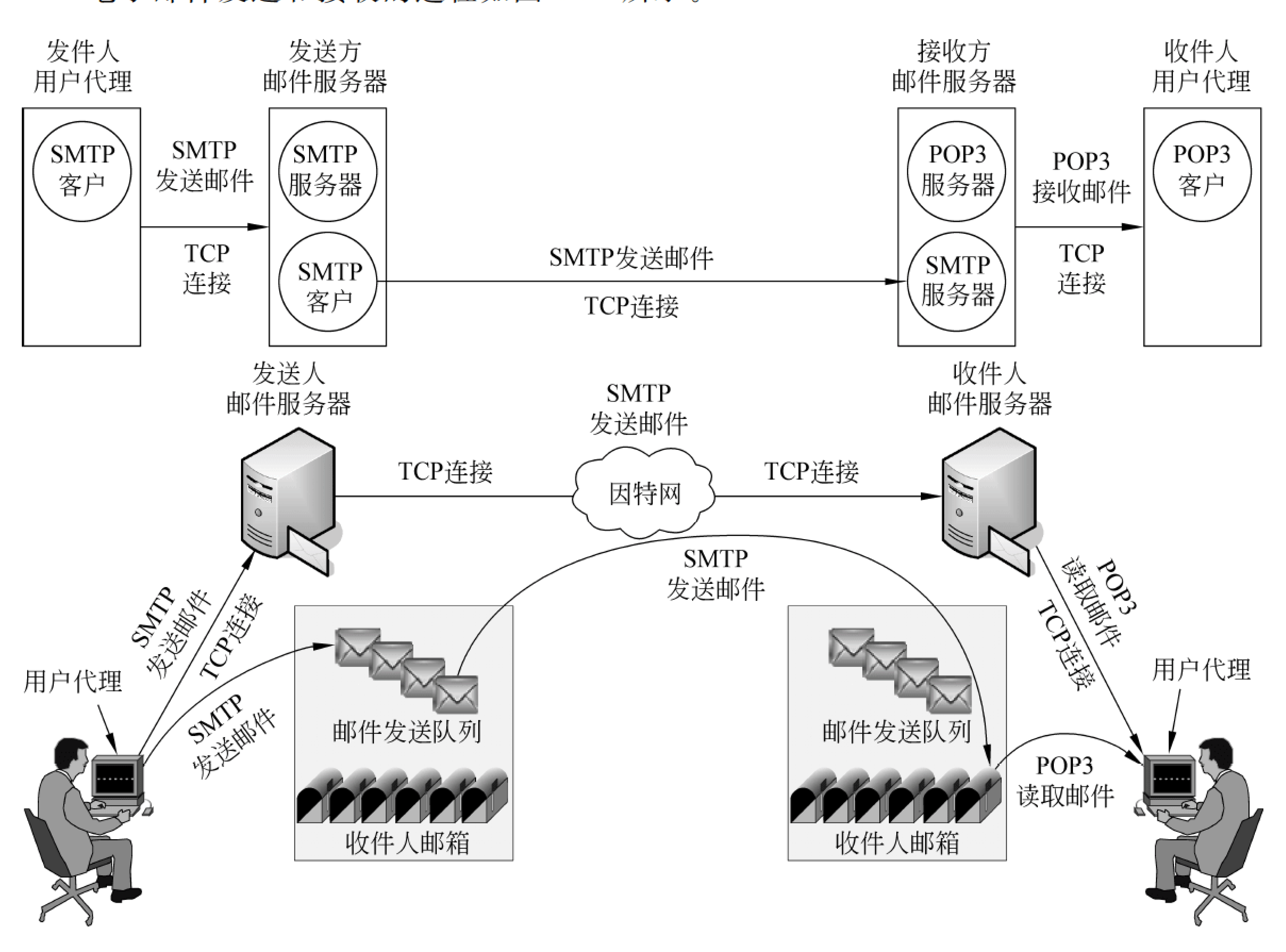


图 4-17 电子邮件系统的组成及工作原理

(1) 发信人使用用户代理编写信件,然后用户代理向发信人的邮件服务器发起 TCP 连接请求。

(2) 当 TCP 连接建立后,用户代理使用 SMTP 协议将邮件传送给发信人的邮件服务

器，然后关闭 TCP 连接。

(3) 发送方邮件服务器将邮件放入邮件发送队列中，等待发送。

(4) 当发送方 SMTP 客户进程发现邮件发送队列中有邮件时，就与收件人的邮件服务器建立 TCP 连接。

(5) 当 TCP 连接建立后，发送方邮件服务器的 SMTP 客户将邮件传送给接收方邮件服务器的 SMTP 服务器，然后关闭 TCP 连接。

(6) 接收方邮件服务器的 SMTP 服务器将接收到的邮件放入收件人的邮箱中（实际是一个用户目录），等待收件人方便时读取。

(7) 收件人收邮件时，运行用户代理，用户代理与收件人邮件服务器建立 TCP 连接。

(8) 当 TCP 连接建立后，用户代理使用 POP3 协议将收件人的邮件从收件人邮件服务器中的邮箱中取回，然后关闭 TCP 连接。

2. 万维网

1) 万维网的组成

在逻辑上，万维网包含 3 个组成要素：浏览器（browser）、Web 服务器（Web server）和超文本传送协议（Hypertext Transfer Protocol，HTTP）。

浏览器在万维网上是与 Web 服务器打交道的客户端程序。浏览器能够按照规定的格式显示 Web 服务器或文件系统内的文档。常见的浏览器有 IE、Chrome、Firefox、Opera 等。

Web 服务器是万维网上提供 Web 服务的程序。当浏览器连接到 Web 服务器并请求某个网页文件时，Web 服务器将根据该请求将网页文件传送给浏览器。Web 服务器使用 HTTP 协议与浏览器进行信息交互，所以它也被称为 HTTP 服务器。Web 服务器不仅能够存储信息，还能根据浏览器提供的信息运行脚本（script）和程序。常见的 Web 服务器有 Apache 和 Microsoft 的 IIS（Internet Information Server）。Web 服务器需运行在一台因特网主机上。

2) 网站、网页与 HTML 语言

万维网是一个分布式信息系统，是由大量的网页（Web page）组成的。网页文件由超文本标记语言（Hypertext Markup Language，HTML）编写，内容包括文字、图片、动画、声音等多种媒体信息以及实现与其他网页、网站或资源的关联和跳转的超链接。网页能被浏览器识别、解释并显示。网页文件本身是一个文本文件，扩展名为 htm 或 html。

具有特定主题的相关网页的集合称为 Web 网站（Website）。网站建立在 Web 服务器上，一个网站上有多少网页没有明确的规定，即使只有一个网页也能称为网站。进入 Web 网站看到的第一个网页称为首页或主页（homepage）。

要制作一个网站，首先需要单独编辑若干个网页文件，然后通过超链接建立起它们之间的逻辑连接关系，并存入 Web 服务器的发布目录，这样网站就建好了。

要访问一个网站，只需要在浏览器的地址栏中输入该网站所驻留的 Web 服务器主机的 IP 地址或域名即可。

【例 4-7】 建立一个简单的网页文档并在浏览器中显示。

在 Windows 中打开记事本，输入以下内容，然后另存为 index.html 文件。双击该文件即可启动浏览器将网页显示在屏幕上。显示的结果如图 4-18 所示。

```
<HTML>
  <HEAD>
    <TITLE>一个简单的网页文档例子</TITLE>
  </HEAD>
  <BODY>
      <H1>这是一个简单的网页文档例子</H1>
      <P>建立：此文件在 Windows 中用记事本建立，文件名为 index.html。</P>
      <P>执行：双击 index.html 文件。</P>
      <P>点击<A href="http://www.baidu.com/">这个链接</A>会打开百度的主页。</P>
  </BODY>
</HTML>
```

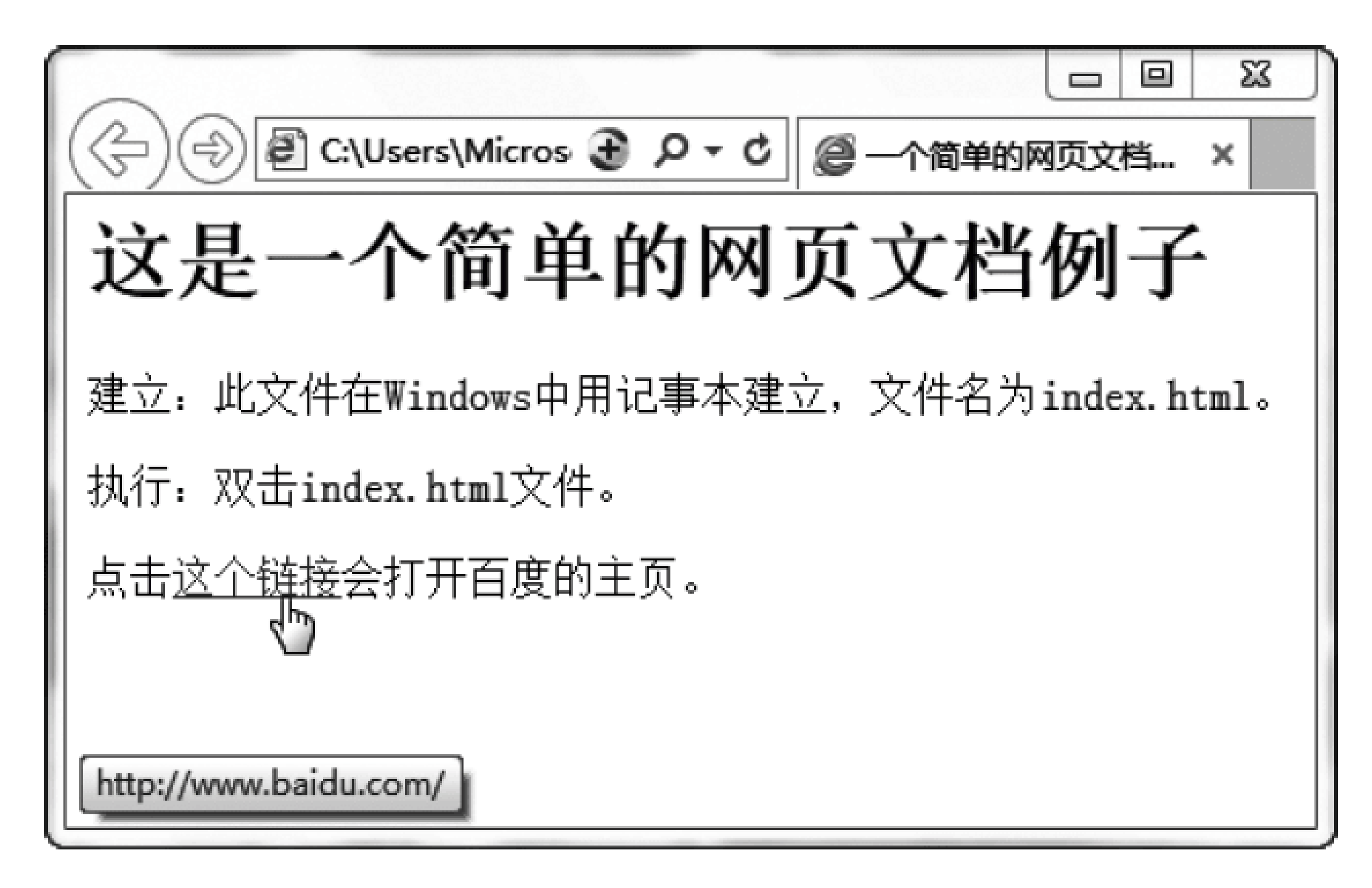


图 4-18 网页文档的显示效果

3）统一资源定位符

统一资源定位符（Uniform Resource Locator，URL）是万维网中指定资源地址的方法。URL 能够使万维网中的所有资源都能用统一的方法进行描述，从而将分散的孤立信息点连接起来，实现资源的统一寻址。这里的“资源”是指因特网中可以被访问的任何对象，包括文件、文件目录、文档、图像、声音、视频等。URL 大致由 3 部分组成：协议、主机名和端口、文件路径。其中对于常用服务，如 WWW、FTP、电子邮件等，端口可以省略。URL 的格式如下：

<协议>://<主机>:<端口>/<路径>

其中，<主机>部分使用域名或 IP 地址均可。<端口>如果不指定，表示使用熟知端口。例如，西安交通大学主页的 URL 表示为 http://www.xjtu.edu.cn/index.html，也可以表示为 http://www.xjtu.edu.cn:80/index.html。

4）超文本传输协议

HTTP 协议是浏览器和 Web 服务器之间用于传输网页文件的应用层协议，它的工

作要依赖于传输层的 TCP 协议。HTTP 是一个面向对象的协议，当一个网页由多个对象构成(如网页中包含的一些图片、视频等对象)时，HTTP 会按每次一个对象进行多次传输，而每次传输都要建立一次 TCP 连接。这种工作方式不仅保证了正确传输网页文件，还能确定每次传输网页中的哪一部分，以及哪部分内容优先显示(如文本先于图形)等。

HTTP 协议定义了浏览器如何向 Web 服务器发出请求以及 Web 服务器如何将 Web 页面返回给浏览器。当用户请求一个 Web 页面时，浏览器发送一个 HTTP 请求报文给 Web 服务器，该 HTTP 请求消息包含了所要求的网页信息。Web 服务器收到请求后，将所请求的网页包含在一个 HTTP 响应报文中，并传送给发出请求的浏览器。HTTP 请求和响应的过程(见图 4-19)可描述如下：

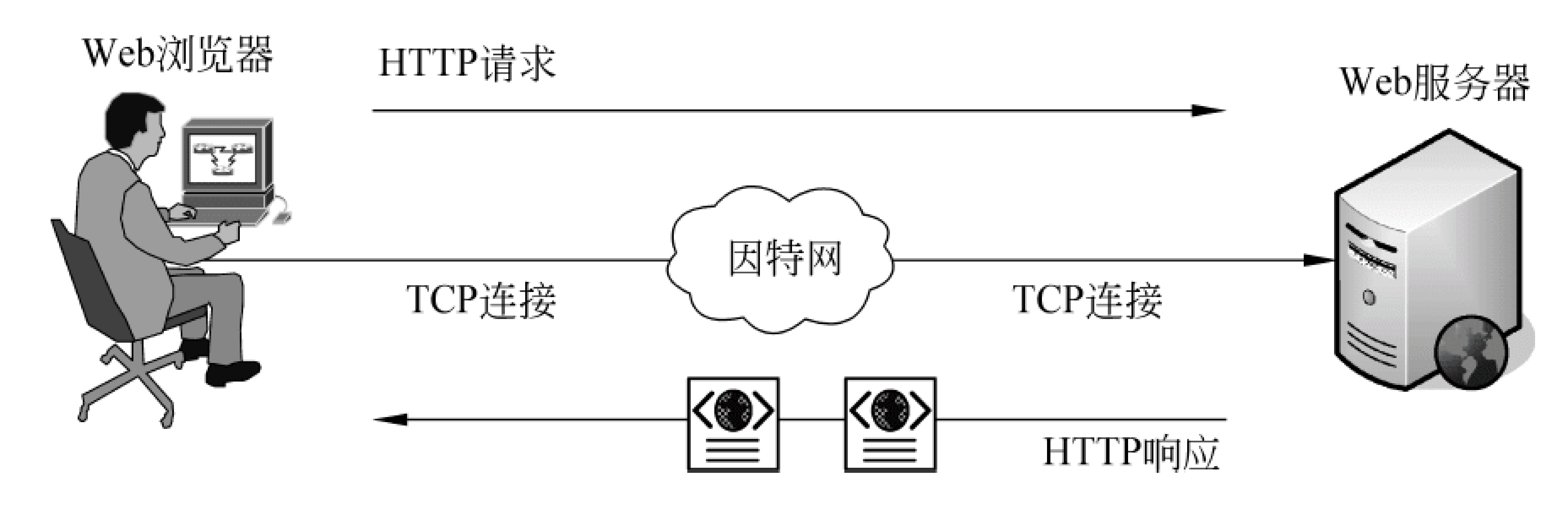


图 4-19　万维网按 B/S 方式传输网页文件

(1) 浏览器分析 URL。

(2) 浏览器向 DNS 请求解析 Web 服务器域名(例如 www.xjtu.edu.cn)的 IP 地址。

(3) 在得到 Web 服务器的 IP 地址后，浏览器与 Web 服务器建立 TCP 连接，默认端口号为 80。

(4) 浏览器通过所建立的 TCP 连接向 Web 服务器发送 HTTP 请求报文，该请求报文中包含了 URL 中网页文档的文件路径名部分(例如/index.html)。

(5) Web 服务器收到请求消息后，从本地读取网页文档并且将其封装到一个 HTTP 响应报文中，然后将 HTTP 响应报文通过 TCP 连接发送给浏览器。

(6) 浏览器接收到响应报文后，释放 TCP 连接。

(7) 浏览器从响应报文中解析出网页文档内容，对其进行解释，然后按规定的格式将网页显示在屏幕上。

如前所述，由于一个网页往往包含多个 HTTP 对象，所以一次会话会重复执行上述过程中的步骤(3)～(7)很多次才能在屏幕上显示完整的页面。

3. BT

BT(BitTorrent)是一个多点下载的开源 P2P 软件，它采用高效的软件分发和点对点技术共享大容量文件，并使每个用户像网络重新分配结点那样提供上传服务。基于 P2P 技术的 BT 系统可以使下载服务器同时处理多个文件的下载请求，而无须占用大量带宽。

一般来讲，FTP、TFTP、HTTP 以及 PUB 这样的下载方式都是基于传统的 C/S 模式进行下载，其基本原理就是将数据放置在一个中央服务器上，需要下载的客户端直接连接

到这个服务器，然后从服务器上读取数据。这种下载方式虽然能够达到下载的目的，但随着用户的增多，对下载带宽的要求也随之增多，同时对服务器的性能要求也会增高，一旦超过一定的阈值，就会产生下载瓶颈，造成服务拥堵，下载速度剧减，甚至会造成中央服务器的临时宕机。所以，在使用传统的下载方式时，很多服务器都会对并发用户人数和下载速度进行一定的限制，这样的处置策略虽然解决了服务器端的瓶颈问题，但明显地给用户造成了很多不便，尤其是在空闲时段不能充分发挥下载性能。

BT 下载采用了与传统下载截然不同的工作机理，如图 4-20 所示。当用 BT 下载一个大文件时，首先把这个文件分解成多个具有特殊标记的分片，然后让不同的 Peer 下载不同的分片到各自的机器上。这样，一个 Peer 所拥有的单个文件，就被分散到其他多个 Peer 上了，但所有这些分散的 Peer 上所存储的都不是完整的初始文件。接下来，在这些分散的 Peer 之间互相分享各自拥有的文件分片，直到每个 Peer 都拥有一个完整的文件为止，至此，整个文件下载过程才算结束。

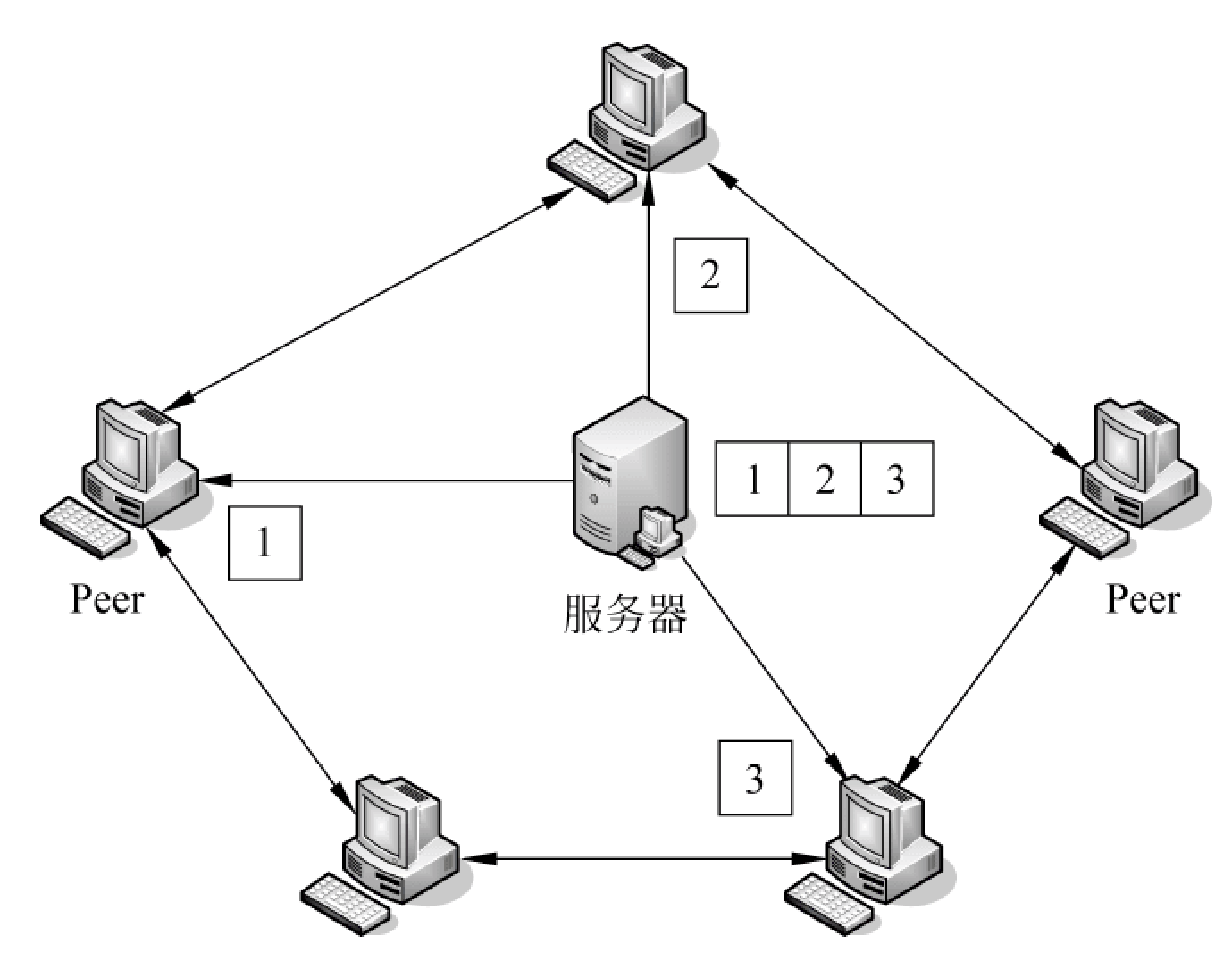


图 4-20　BT 下载原理

从图 4-20 可以看出，作为服务器的 Peer 把文件传给其他的 Peer 后，各个 Peer 之间也可以直接交互传输。这样，单一的下载源就变成了多个下载源，Peer 越多，则彼此之间交互的范围和数量也就越大，相应地，下载速度也就变得越快了。

为了更好地说明 BT 下载过程，接下来再通过一个实例予以说明。假定结点甲、乙、丙希望从服务器上下载同一文件，BT 下载过程可以分为如下 3 个步骤。

(1) 服务器将文件分成 3 个部分，甲下载了第一部分，乙下载了第二部分，丙下载了第三部分。

(2) 甲下载完第一部分后，就可以脱离与服务器的交互，直接与乙和丙结点连接，从乙结点下载第二部分，从丙结点下载第三部分。当甲同时下载完这三部分后，就可以将这三部分进行组合，形成一个完整的文件。

(3) 依次类推，乙可以从甲结点获得第一部分内容，从丙结点获得第三部分内容；而丙则可以从甲结点获取第一部分，从乙结点获取第二部分。最终实现整个文件的下载。

4. 即时通信

即时通信(Instant Messaging,IM)是指能够即时发送和接收因特网消息的业务。利用即时通信工具,用户之间可以实现文字、语音、视频的实时互通交流。现在,即时通信不仅可以提供聊天功能,还集成了电子邮件、文件传输、博客、音乐、视频、游戏和搜索等多种功能。常见的即时通信软件包括 QQ、飞信、Skype、微信等。

即时通信系统最初只是一个 C/S 结构的因特网应用,但现在它已经将 P2P 和 C/S 进行了结合。大多数情况下,用户首先以 C/S 方式从即时通信服务器上获取好友列表,然后用户与好友之间采用 P2P 方式进行通信。当好友之间无法建立 P2P 连接时,则采用服务器中转的方式进行通信,如图 4-21 所示。在这种系统中,即时通信服务器的主要功能是向用户提供好友目录服务,因此使用即时通信服务的用户首先要在即时通信服务器上注册,这样通信各方才能相互定位。

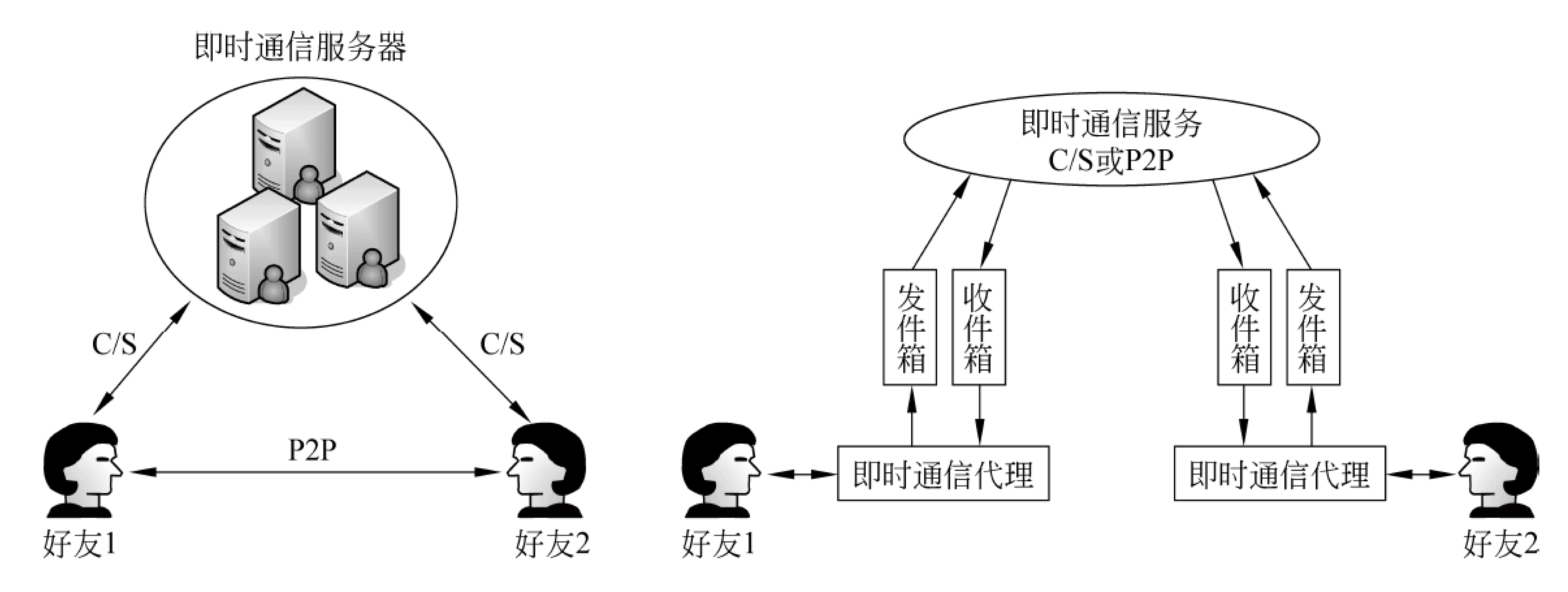


图 4-21 典型的即时通信系统结构和系统模型

即时通信系统虽然也有收件箱/发件箱(有点类似于电子邮件系统),但与电子邮件不同的是,即时消息足够短小,便于快速投递到收件箱。另外,考虑到在线人数非常多,即时通信系统往往采用多服务器组成的集群系统来提供即时通信服务。

即时通信使用的传输层协议为 TCP 协议和 UDP 协议。一般情况下,注册和好友目录服务使用 TCP 协议,而好友之间的通信采用 UDP 协议。例如,QQ 在默认状态下优先采用 UDP 协议进行通信,其原因是 UDP 协议适用于无须应答、注重时效的网络应用,这个特点正好与 QQ 追求的目标相符。

5. 流媒体

传统的多媒体播放方式是先将整个媒体文件内容下载到本地,然后再播放。由于多媒体文件通常容量很大,因此下载需要很长时间。而流媒体采用了流式传输技术,将多媒体文件经特定压缩方式处理成多个压缩包,由视频服务器向用户顺序地实时传送,用户不必等整个文件下载完毕。传统流媒体服务都是基于 C/S 模式的,即用户从流媒体服务器点击观看节目,然后流媒体服务器以单播方式把媒体流推送给用户。这种 C/S 模式加单

播方式的缺陷在流媒体业务发展到一定阶段,用户数量上升到一定规模后就会凸显出来,一些应用商开始尝试在流媒体领域引入 P2P 技术,以便解决流媒体服务器占用带宽大、负载过重等问题。

基于 P2P 技术的流媒体发布具有下列优势:

(1) 提高流服务能力。P2P 流媒体技术可以在核心结点根据 P2P 协议对流媒体内容做切片处理,P2P 用户根据协议规则来完成内容共享。P2P 用户在边缘层的引入大大降低了边缘服务器的压力,提高了流媒体的传输效率。与此同时,P2P 流媒体技术充分利用了用户的闲置上行带宽,这样运营商可以通过更少的边缘服务器提供更多的业务量,为更多的用户服务,以较低成本代价应对迅猛增长的客户规模带来的挑战。

(2) 改善客户体验。P2P 和流媒体的结合使得有限的服务能力可以为更多的用户提供流媒体服务,同时也能够更有效地防止因网络抖动而对服务质量产生影响。另外,内容丰富也是 P2P 流媒体的一大特点,直播、点播、录播等播放方式种类齐全;不仅央视、省级卫视的众多电视频道可以实时或延时收看,还可提供其他经典节目的点播。

当前的 P2P 流媒体技术主要应用在视频点播(VOD)、视频广播(直播)、交互式网络电视(IPTV)、远程教学以及交互游戏等领域,在 P2P 流媒体发展及应用过程中,出现了大批经典的应用系统,除最早的一批以 PPLive、PPStream 为代表的 P2P 流媒体之外,大量的 P2P 网站、P2P 网络电视、P2P 视频点播系统等相继诞生,出现了 QQLive、UUSee、PPMate、SopeCast 等知名应用。

PPLive 网络电视是一款用于互联网上大规模视频直播的免费共享软件,内核采用了独特的应用层多播和内聚算法技术,降低了视频传输对运营商主干网的冲击,减少了出口带宽流量,并能够实现用户越多播放越流畅的特性,有效解决了当前网络视频点播服务的带宽不足和负载有限问题,使得业务的整体服务质量大大提高。

QQLive 是由腾讯公司自主研发的 P2P 流媒体互动传播平台,能够为互联网用户提供稳定和流畅的音视频直播节目。QQLive 采用了 P2P-Streaming 技术,具有用户越多播放越稳定,支持百万级用户同时在线的大规模访问等特点。QQLive 客户端可以应用于网页、桌面程序等多种环境,并提供丰富的视频节目及实时互动功能,能够满足不同类型的用户需求。

PPMate 网络电视是一款基于 Internet 的大规模视频直播软件,也是一款免费绿色的视频内容聚合客户端。PPMate 提供的内容均是程序在网络上自动收集的,版权归内容所有者,它只是为网友提供视频交流平台,不存储任何视频内容。PPMate 的节目列表由 PPMate 爬虫程序自动采集网页生成,不能保证所提供的文字描述和实际内容一定相符。

UUSee 网络电视是悠视网打造的一款全新的网络电视收看软件,用户使用这款软件可以免费收看 500 多路新颖频道,共计 1000 多个精彩节目。UUSee 的主要功能特点包括多模式播放、高清晰视频、酷热节目内容、快速录制、准确节目预告、强劲视频流搜索、个性化频道管理等。

4.3 局　域　网

大多数人初次上网接触到的是局域网,大多数因特网中的主机也都是通过局域网接入的,特别是企业、政府部门、学校和研究机构等企事业单位中的主机。即便是家庭用户用宽带上网也不可避免地涉及局域网连接。典型的局域网是以太网(Ethernet)和IEEE 802.3局域网,这两种局域网几乎没有什么区别,对一般用户来说可以等同看待。

4.3.1 局域网结构和标准

1. 结构

图4-22是一个简单局域网的结构示意图,其主要部件如下:

- 主机设备,包括计算机、服务器等。
- 网络设备,包括网络接口卡、交换机、路由器/网关、无线接入点等。
- 传输介质,包括双绞线、光纤、无线介质(如WLAN)等。

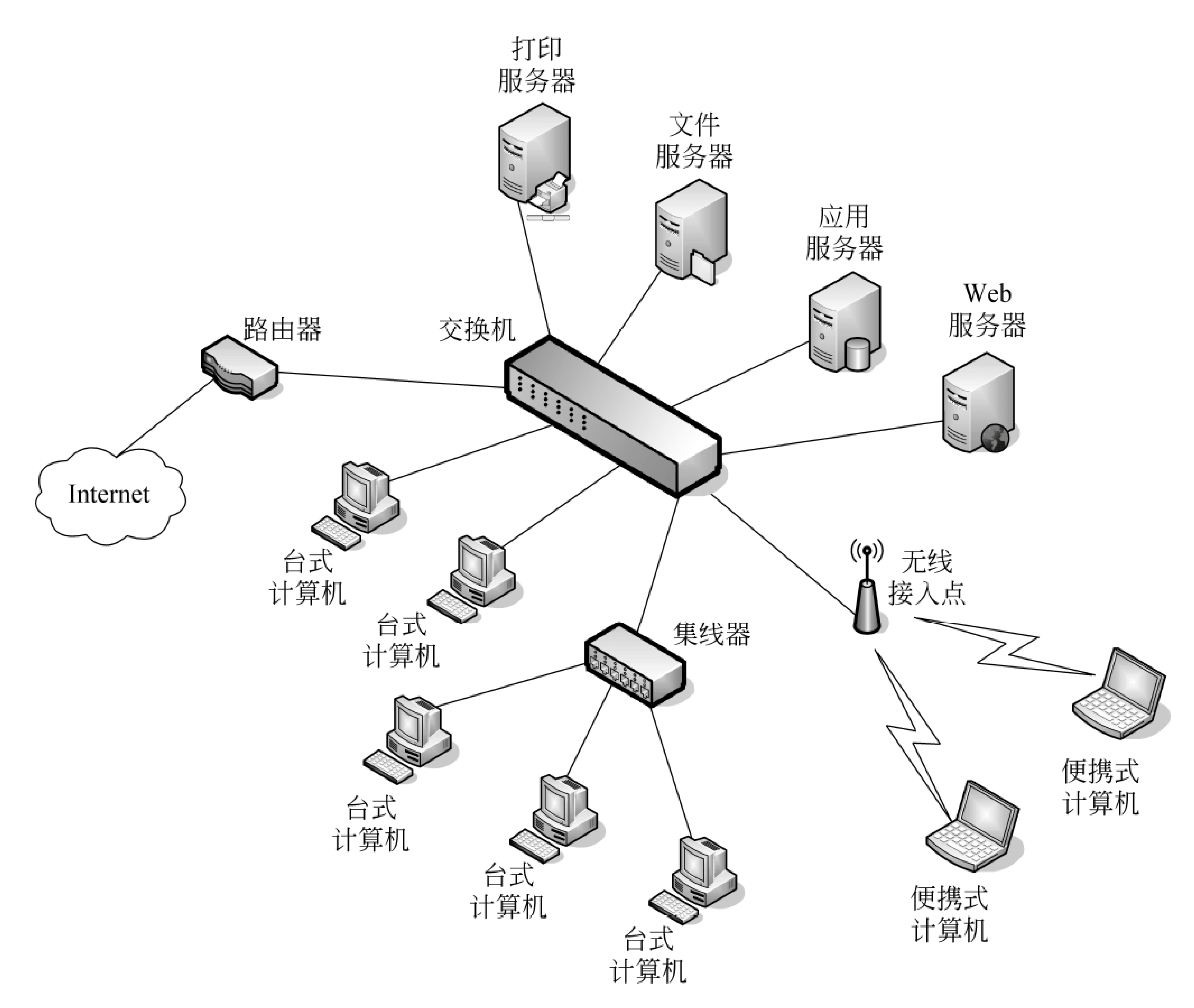


图4-22　一个简单的局域网结构示意图

从网络体系结构上观察,局域网技术只涉及OSI/RM定义的7层中的最低两层,即

数据链路层和物理层。在局域网中，当数据链路层收到其上层——网络层递交的 IP 分组时，就将其封装到本层的 PDU——帧(Frame)中，然后交由物理层进行传输。反之，当物理层从传输介质上接收到一个完整帧时，它就将这个帧提交给数据链路层，然后由数据链路层从帧中剥离出 IP 分组提交给网络层进行处理。

星形拓扑结构在现代局域网中应用较多，特别是集线器(hub)和交换机(switch)在局域网中的大量使用，使得星(树)形结构成为局域网的主流结构。总线拓扑也是局域网中非常流行的一种拓扑形式，具有可靠性高、扩充方便的优点，曾广泛应用于以太网中，但目前已被星形结构所替代。环形结构只出现在早期的令牌环网中，现在已不多见。

2. 标准

IEEE 802 系列标准(由 IEEE 802 委员会制定的局域网标准，简称 802 标准)是最成熟、标准化程度最高的局域网标准。802 标准不仅已被美国国家标准局(ANSI)接纳为美国国家标准，还被 ISO 正式接纳为国际标准(ISO 8802)，从而在世界范围得到了广泛应用。802 标准完全遵循了 OSI 参考模型的原则。它主要描述了网络体系结构中最低两层——物理层和数据链路层的功能。

802 标准的物理层规定了物理传输所使用的信号编码和介质，规定了网络的拓扑结构和传输速率。802 标准的数据链路层分为逻辑链路控制(Logical Link Control，LLC)子层和介质访问控制(Media Access Control，MAC)两个功能子层。这两个子层将数据链路功能中与硬件相关的部分和与硬件无关的部分分离开来，从而使局域网体系结构在 LLC 子层不变的条件下，只需更换 MAC 子层便可适应不同的传输介质和介质访问控制方法。这是网络体系结构分层思想在局域网中成功应用的一个典型案例。

802 标准由一系列的子标准组成，并且还在不断扩充。其中主要的子标准包括：

- IEEE 802.2 逻辑链路控制。
- IEEE 802.3 CSMA/CD 总线访问控制及物理层规范(以太网)。
- IEEE 802.11 无线局域网访问控制及物理层规范(WLAN)。

4.3.2 局域网设备

构建局域网所需的设备主要有网络接口卡、网络交换机和网关，如果要构建无线局域网，则需要无线网卡、无线接入点(Access Point，AP)或无线路由器(AP、交换机和路由器三合一设备)。

1. 网络接口卡

网络接口卡(Network Interface Card，NIC)简称网卡，又称网络适配器。早期的网卡是插在计算机总线插槽内或 USB 接口上的扩展卡，称为独立网卡。现在的计算机主板上大多数已经集成了网络接口，称为集成网卡。在局域网中，每台计算机至少应具有一个网络接口(有线或无线)。

2. 网络交换机

网络交换机是一种集中连接设备，如图 4-23 所示。利用它可轻松地组建以双绞线/光纤为介质的星形结构局域网。交换机工作在数据链路层，能够根据 MAC 地址将帧转发到合适的连接端口。

用网络交换机组网还有一个很大的优点，交换机的每个接口都是一个独立的冲突域，因此它能够隔离冲突域，降低接口所连接的网段的冲突概率。



图 4-23 网络交换机(Netgear 12-port 10/100/1000Mbps Managed Fiber Gigabit Switch)

3. 网关

局域网中的网关(gateway)本质上是一台路由器，是局域网通向因特网的关口，用于在局域网与因特网之间转发分组。当局域网中的主机要与因特网中的其他主机进行通信时，必须要通过网关。图 4-22 中与因特网相连的路由器就是该局域网的网关。

局域网可以设置多个网关，主机可以指定 IP 分组从哪个网关发到因特网。如果局域网中只有一个网关，则可以在主机中将这个唯一的网关设置为默认网关(也称默认网关)。主机如果找不到可用的网关，就把 IP 分组发往默认网关，由默认网关来转发 IP 分组。在主机中，默认网关是不能随便指定的，如果设置不正确，IP 分组就会传送到不是网关的主机，从而无法访问因特网。常见的 ARP 病毒就是通过非法修改主机上的网关设置，造成局域网中的主机无法访问因特网的故障。

4. 无线接入点

无线接入点是无线局域网中的"无线基站"，类似于有线局域网中的集线器，用于为无线站点之间或无线站点与有线局域网之间提供通信转接能力。在一个具有 AP 的 WLAN 中，无线站点可以直接与另一个无线站点通信，也可以通过 AP 与另一个无线站点通信(此时 AP 的作用是负责站点之间的信息转发)。如果把无线接入点 AP 与有线局域网连接，它还可以用作 WLAN 和有线局域网之间的桥接器，将多个无线站点接入到有线局域网上。

在构建家庭网络和 SOHO(Small Office & Home Office)网络时，还经常用到一种称为无线路由器(wireless router)的设备(参见图 4-24)。无线路由器将广域网接口、无线网络接口和以太网交换机集成在一起，既有路由器的功能，又有 AP 的功能，还有网络交换机的功能(通常有 4 个有线以太网接口)，这就为构建小型有线/无线混合型网络带来了极大的方便。



图 4-24 无线路由器(AP+路由器+交换机)

在网络的逻辑结构上，通常把无线路由器当作 AP 看待。

*4.4 网 络 安 全

本节主要关注的是互联网安全，包括互联网网络环境下计算机系统和互联网自身的安全保护，以及基于互联网的通信和分布式应用系统的安全保护。

4.4.1 网络安全概念

1. 信息系统安全

在信息时代，信息、计算机和网络是不可分割的整体。因此，信息系统安全包括了信息安全、计算机安全和网络安全。

信息安全是指信息内容的安全。保护信息的真实性、保密性和完整性，避免攻击者利用系统的安全漏洞进行窃听、诈骗等危害合法用户利益的行为。涉及信息基础设施（各种通信设备、信道、终端和软件等）、信息资源和信息管理。

计算机安全是指“为数据处理系统建立和采取的技术和管理的安全保护，保护计算机硬件、软件和数据不因偶然和恶意的原因而遭到破坏、更改和泄密”。涉及物理安全和逻辑安全两个方面的内容。物理安全指计算机系统设备及相关设备的安全，逻辑安全则指保障计算机信息系统的安全，即保障计算机中信息的完整性、保密性和可用性。

网络安全是指网络上的信息安全，主要指网络系统的硬件、软件及其系统中的数据受到保护，不受偶然的或者恶意的原因而遭到破坏、更改、泄露，系统连续、可靠、正常地运行，网络服务不中断。

2. 安全风险

信息系统由硬件设备、系统软件、数据资源、服务功能和用户等基本元素组成，与之相关的安全风险因素包括自然灾害威胁、系统故障、操作失误和人为蓄意破坏。而这些不安全因素则是由信息系统本身的脆弱性所决定的。

网络的开放性使得网络系统的协议、核心模块和实现技术是公开的，其中的设计缺陷很可能被别有用心的人所利用。网络的全球化可以使攻击者实施对网络的远程攻击。基于网络的各成员之间的信任关系可能被假冒。

由于网络的开放性和网络技术的普及，因特网上存在大量公开的黑客站点，获得黑客工具、掌握黑客技术越来越容易，从而导致信息系统所面临的威胁日益严重。

攻击者通过多种手段实现自己的非法目的。从攻击的方式区分，可以分为被动攻击和主动攻击两种，其中被动攻击的主要目标是进行信息收集并从通信流量的特征中寻找有用的、有利可图的信息，攻击者通过 sniffer、wiretapping 和 interception 等工具进行窃听，然后进行流量分析来达到自己的目的；主动攻击与被动攻击不同，它包含主动的访问

行为，具体方式包括以下 7 种类型：

- 阻断（interception）：切断通信路径或端系统，破坏网络和系统的可用性。
- 篡改（modification）：未经授权修改信息，破坏系统或数据的完整性。
- 重放（replay）：捕获通信路径中的数据单元，在以后的某个时机重传。
- 伪造（fabrication）：假冒另一个实体发送信息。
- 拒绝服务（denial of service）：通过耗尽目标系统的资源以危害目标系统的正常使用。
- 恶意代码（malicious mobile code）：如病毒（virus）、蠕虫（worm）、木马（trojan）、恶意脚本（JavaScript、Java Applet、ActiveX）等。
- 抵赖（repudiation）：包括源发抵赖和交付抵赖。

计算机病毒（computer virus）是一种人为编制出来嵌入到其他软件中的程序段。它具有感染性、隐蔽性、潜伏性、破坏性、可触发性、攻击的主动性、不可预见性等特征。木马也属于一种计算机病毒。“木马”这种称谓是借用于古希腊传说中的著名计策——木马计。木马程序与一般的病毒程序不同，它不会自我繁殖，也并不“刻意”地去感染其他文件，它是通过将自身伪装成一个非常吸引人们眼球的应用（如一个好玩的小游戏、一段视频、一个非常稀罕的资源等），吸引用户下载或执行，这时木马就会以插件形式嵌入到用户主机的操作系统、浏览器等软件中，在受害者计算机中打开了一个后门，使施种者可以任意毁坏、窃取被害者的敏感信息（如银行账号、密码等），甚至远程操控被害者的计算机去做危害网络的操作。

3. 安全目标、服务和机制

网络的安全目标、安全服务和安全机制之间的关系如图 4-25 所示。网络通过各种安全服务实现网络的保密性、完整性和可用性等安全目标；安全服务是安全系统的功能体现；而安全机制则是实现安全服务的保证。一种安全服务可以由多种安全机制实现；同时，一种安全机制也可用于实现多种安全服务。

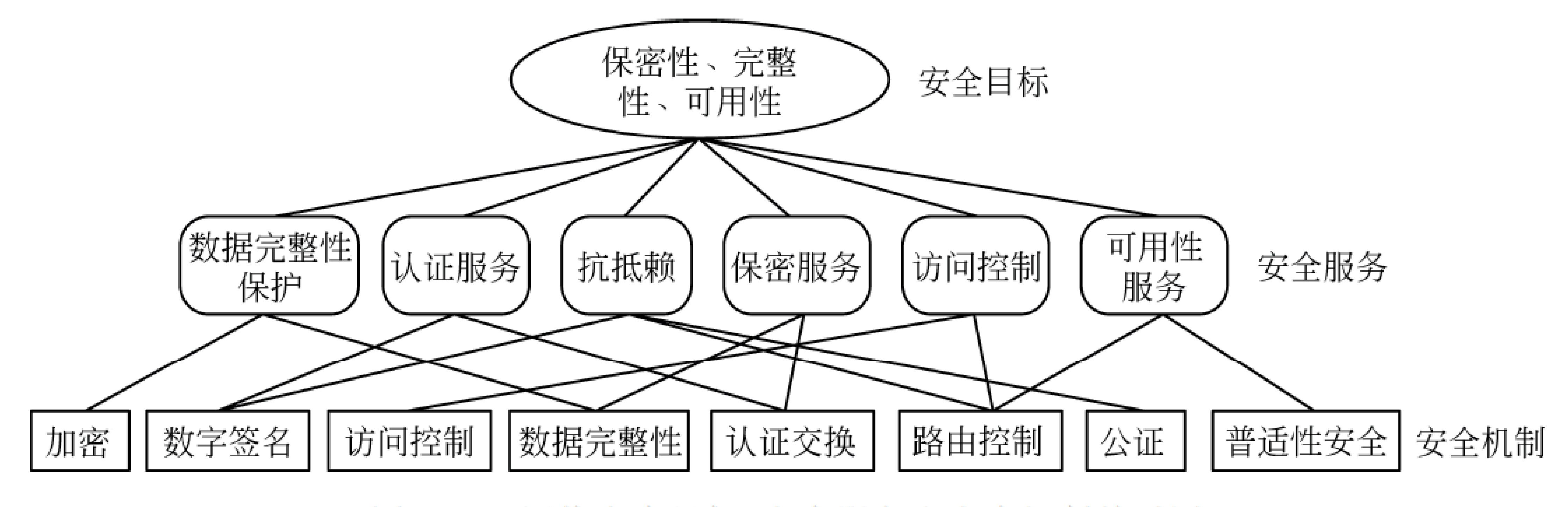


图 4-25　网络安全目标、安全服务和安全机制关系图

网络安全的本质目的就是保护网络信息的保密性、完整性和可用性。

- 保密性（confidentiality）。也称为机密性，是指阻止非授权的被动攻击，保护网络中的信息内容包括业务数据、网络拓扑、流量特征等不会被泄露给未授权的实体。

- 完整性(integrity)。主要针对主动攻击而言,指保证信息不被未经授权地篡改,或者能够保证检测出被修改的内容。
- 可用性(availability)。是指防止对计算机系统可用性的攻击(拒绝服务攻击),保证资源的授权用户能够访问到应得资源或服务,如路由交换设备的分组处理能力、缓冲区、链路带宽等。

网络安全服务是指计算机网络提供的安全防护措施。国际标准化组织(ISO)定义了以下几种基本的安全服务:

- 认证服务。包括对等实体认证和数据源发认证。前者是面向连接的应用,目的是确保参与通信的实体身份真实。后者面向无连接的应用,目的是验证收到的信息的确来自它所宣称的来源。
- 保密服务。分为连接保密服务与无连接保密服务。保密服务主要进行信息流的保密。保密粒度分为流、消息和选择字段等。
- 数据完整性保护。同保密服务类似,分为面向连接和无连接的完整性保护。保护粒度也分为流、消息和选择字段等。数据完整性包括以下两种实现方式:访问控制方式(未授权者无法修改信息)和验证码方式(通过消息验证码实现数据完整性保护,消息被未授权的修改可以被检查出来)。
- 访问控制。指通过不同的授权限制实体的访问权限。访问控制实现的前提是标识与认证。
- 可用性服务。指通过资源冗余(备份)防止针对计算机系统可用性的攻击,以及用于灾难恢复。
- 抗抵赖。指通过有效的措施和机制(如数字签名)防止用户否认其行为(如已发送的消息)。包括发送抗抵赖和交付抗抵赖。

网络安全机制是用于实现安全服务的机制。安全机制既可以是具体的、特定的,也可以是通用的,主要的安全机制有以下几种:

- 加密机制。又称为密码机制,用于支持数据保密性、完整性等安全服务。加密机制的算法可以是可逆的,也可以是不可逆的。
- 数字签名机制。包括签名和验证。签名应采用签名者独有的私有信息。验证则应使用公开的信息和规程。
- 访问控制机制。根据事先确定的规则检测主体访问客体的合法性及权限。访问控制可以基于多种手段进行,例如集中的授权信息库、主体的能力表、客体的访问控制链表、主体和客体的安全标签或安全级别以及路由、时间、位置等。访问控制的位置可以在源点、中间或目的结点。
- 数据完整性机制。包括单个数据单元的完整性以及数据单元序列的完整性。前者主要通过添加标记进行检测,后者主要通过添加序列号和时间戳等进行检测。
- 认证交换机制。用交换信息的方式来确定身份的技术。交换的内容包括认证信息,如口令、密码技术、被认证实体的特征等。为防止重放攻击,常与时间戳、两次或三次握手、数字签名等机制结合使用。
- 路由控制机制。动态地或根据事先预设的方式选择路由,以确保只使用物理安全

的子网、中继站或链路。例如，在检测到持续的操作攻击时，端系统可指示网络服务提供者经不同的路由建立连接；带有某些安全标记的数据可能被安全策略禁止通过某些子网、中继或链路，此时，连接的发起者（或无连接数据单元的发送者）可以指定路由选择，请求回避这些特定的子网络、链路或中继。

- 公证机制。确保在两个或多个实体之间的可靠身份、通信数据的性质（如完整性、源发、时间和目的地等）的机制。公证机制由通信实体都信任的第三方实体——公证机构提供，公证机构须掌握必要信息以确保提供所需的公证服务。
- 普适性安全机制。包括安全标签、事件检测、审计跟踪和安全恢复等。

4.4.2　密码学基础及应用

密码学是研究信息系统安全的学科，它分为两个分支，即密码编码学和密码分析学。密码编码学是密码体制的设计学，而密码分析学则是在未知密码的情况下从密文推演出明文或密钥的技术。密码学作为保护信息的手段，经历了从古典密码学到现代密码学的转变。古典密码学的算法主要是通过字符之间代替或易位实现的，包括单表代替密码、多表代替密码等。尽管这些密码算法大都十分简单，破解也相对容易，但对于现代密码学的发展和进步也有很大的参考意义。现代密码学与计算机、通信网络的广泛应用密切相关，它不仅要提供古典密码学所解决的机密性的手段，而且要提供信源、数据的真实性和完整性的方法。

1. 数据加密

数据加密是以某种特殊的算法改变原有数据的表现形式，使得未授权的用户即使获得了已加密的信息，但因不知解密的方法，仍然无法了解信息的内容。由于网络的开放性，黑客很容易截获网络上传输的数据，这就使加密技术成为保障网络安全的技术手段之一。

任何一个加密系统（密码系统）都是由明文、密文、算法和密钥组成的，如图 4-26 所示。其中明文就是原始数据；密文是加密后的数据；算法是加密过程中所采用的变换方法，如加密算法是将明文转换为密文，解密算法是对密文实施与加密相逆的变换，获得明文的过程；密钥是事先规定好的用于对明文进行加密、对密文进行解密的特殊信息。

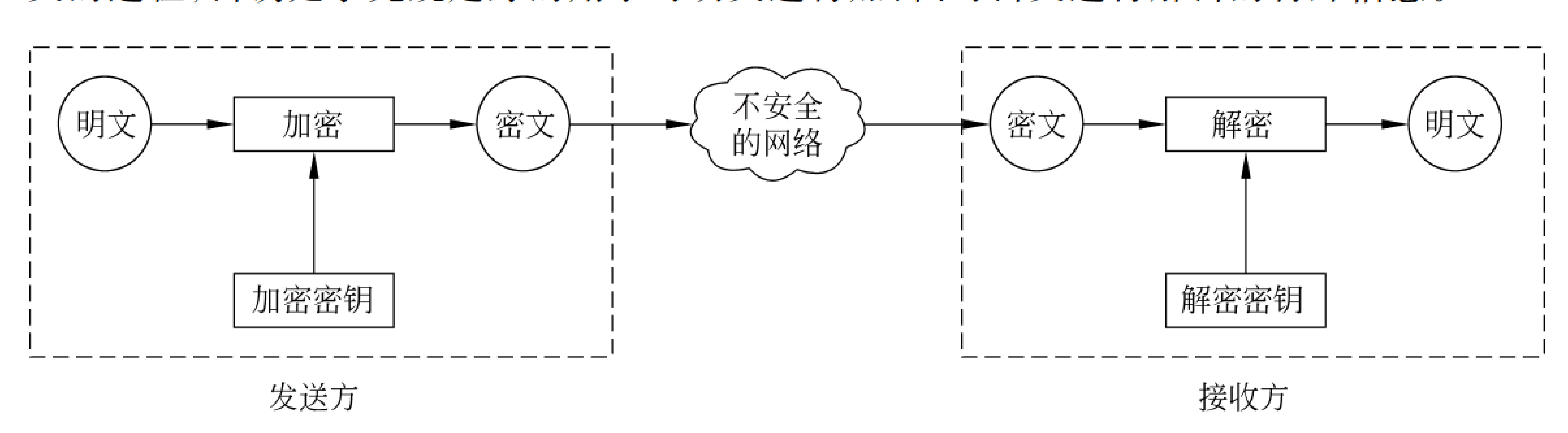


图 4-26　加密和解密过程示意图

发送方使用加密算法，用加密密钥将明文转换成密文后发送出去。接收方收到密文后，用解密密钥将密文转换为明文。在传输过程中，即使密文被攻击者窃取，得到也只是无法识别的密文，从而起到信息保密的作用。

1）传统加密算法

传统的加密算法基本可以分成四类。

（1）单字符替代加密。

在替代密码中，用一组密文字母来代替一组明文字母以隐藏明文，但保持明文字母的位置不变。单字符替换加密(mono alphabetic cipher)是一种简单的替代加密法，最古老的单字符替代密码是凯撒密码(Caesar ciper)，它用 d 表示 a，用 e 表示 b，用 f 表示 c……用 c 表示 z，也就是说密文字母相对明文字母右移了 3 位。

【例 4-8】 明文是 Cipher 的凯撒密文是什么？

密文是“Flshu”。更一般地，可以让密文字母相对明文字母右移 k 位，这样 k 就成了加密和解密的密钥。这种密码是很容易破译的，因为最多只需尝试 25 次($k=1\sim25$)即可轻松破译密码。

单字符替代加密的主要特征是：密文中的同一个字符去代替明文中的对应字符，破译者只要拥有很少一点密文，利用自然语言的统计特征，很容易就可破译密码。破译的关键在于找出各种字母或字母组合出现的频率。比如经统计发现，英文中字母 e 出现的频率最高，其次是 t、o、a、n、i 等，最常见的两字母组合依次为 th、in、er、re 和 an，最常见的三字母组合依次为 the、ing、and 和 ion。因此破译者首先可将密文中出现频率最高的字母定为 e，频率次高的字母定为 t……然后猜测最常见的两字母组、三字母组，比如密文中经常出现 tXe，就可以推测 X 很可能就是 h，如经常出现 thYt，则 Y 很可能就是 a 等。同时，单字符替代加密并没有隐藏原来单词的长度，因此可以猜测这些加密的单词，如一个字母的单词可能是 a 或 I，两个字母的单词可能是 or、is、an、it 或 on，三个字母的单词可能是 and 或 the。采用这种合理的推测，破译者就可以逐字逐句组织出一个试验性的明文。

（2）多字符替代加密。

为了去除密文中字母出现的频率特征，可以使用多字符替代加密，使明文字母和密文字母之间的映射关系没有固定规律，即明文中的同一字母不总是被密文中的固定字母所替换。多字符替换加密的一个重要例子是 Vigenere 密码，需要一个密钥和一个 Vigenere 表，如表 4-5 所示。加密时，把密文周期性地写在明文上方。用明文中的一个字母对应表的列，该字母上方的密文字母对应表的行，这样就可以从 Vigenere 表中查到加密后的替代字母。

【例 4-9】 如果明文是“I LOVE STUDYING NETWORKING”，密钥是“YOUMUSTBECRAZY”，则使用 Vigenere 表加密后的密文是什么？

密文如表 4-6 所示，第一个密码由 Vigenere 表的第 Y 行第 I 列确定，第二个密码 Z 由 Vigenere 表的第 O 行第 L 列确定，以此类推。

表 4-5　Vigenere 表

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

表 4-6　例 4-9 加密过程

Y	O	U	M	U	S	T	B	E	C	R	A	Z	Y	Y	O	U	M	U	S	T	B	E	密钥
I	L	O	V	E	S	T	U	D	Y	I	N	G	N	E	T	W	O	R	K	I	N	G	明文
G	Z	I	H	Y	K	M	V	H	A	Z	N	F	L	C	H	G	A	L	C	B	O	K	密文

另一种演变的多字符替代加密法是采用多张密码字母表，比如任意选择 26 张不同的单字母密码表，相互间排定一个顺序，然后选择一个简短易记的单词或短语作为密钥，在加密一条明文时，将密钥重复写在明文的上面，则每个明文字母上的密钥字母即指出该明文字母用哪一张单字母密码表来加密。

【例 4-10】　如果明文是 please execute the latest scheme，密钥是 computer，采用多字母密码表时，对应密文是什么？

密文如表 4-7 所示。这里假设 a～z 分别表示顺序 1～26，则第 1 个明文字母 p 用第 3 张单字母密码表加密，第 2 个明文字母 l 用第 12 张单字母密码表加密……显然，同一个明文字母因位置不同而在密文中可能用不同的字母来表示，从而消除了各种字母出现的频率特征。

表 4-7　例 4-10 加密过程

c	o	m	p	u	t	e	r	c	o	m	p	u	t	…	e	r	c	o	m	p
p	l	e	a	s	e	e	x	e	c	u	t	e	t	…	s	c	h	e	m	e

虽然破译多字母密码表要困难一些，但如果破译者手头有较多的密文，仍然是可以破译的，破译的诀窍在于猜测密钥的长度。首先破译者假设密钥的长度为 k，然后将密文按每行 k 个字母排成若干行，如果猜测正确，那么同一列的密文字母应是用同一单字母密码表加密的，因此同一列中各密文字母的频率分布应与英文相同，即最常用的字母（对应明文字母 e）频率为 13％，次常用的字母（对应明文字母 t）频率为 9％，等等。如果猜测不正确，则换一个 k 进行重试，一旦猜测正确，即可逐列使用破译单字母表密码的方法进行破译。进一步提高破译难度可以使用比明文还长的密钥，使上述破译方法失效，但这样的密钥难以记忆，如果记在本子上又会增加失密的可能性。

(3) 换位加密。

换位加密（transposition cipher）是将明文字母的次序进行重新排列，并不对明文字母进行变换。最简单的换位加密是把要处理的明文填入一个表中，然后按照另一种方式读该表，形成密文。

【例 4-11】 将明文“company results are as expected”按照表 4-8 填入后的换位密码有哪些？

表 4-8　换位加密表

c	o	m	p	a
n	y		r	e
s	u	l	t	s
	a	r	e	
a	s		e	x
p	e	c	t	e
d				

按列输出的密文是 cns apd oyuase m lr c prteet aes xe。如果使用一个密码指定列的转换次序，如 24351，表示先转换第 2 列，然后第 4 列，以此类推。按列输出的密文是 oyuase prteet m lr c aes xe cns apd。

换位密码并不安全，它仍然保留了字母的频率信息，破译者可以尝试将这些字母重组为各种大小的矩阵，观察出现的单词。当然，如果把明文按照对角线或螺旋形填表，或者对密文再用第二张表进行一次转换，可以使密文更复杂。

另一种演变的换位加密法是在加密时将明文按照密钥长度截成若干行排在密钥下面，密钥必须是一个不含重复字母的单词或短语，按照密钥字母在英文字母表中的先后顺序给各列进行编号，然后依照序号顺序按列输出密文。

【例 4-12】 如果明文是 pleaseexecutethelatestScheme，密钥是 COMPUTER，采用换位加密后的密文是什么？

按照表 4-9，该明文的密文是 pelhehsceutmlcaeateexecdettbsesa。

表 4-9　例 4-12 加密表

C	O	M	P	U	T	E	R
1	4	3	5	8	7	2	6
p	l	e	a	s	e	e	x
e	c	u	t	e	t	h	e
l	a	t	e	s	t	s	c
h	e	m	e	a	b	c	d

破译的第一步是判断密码类型，检查密文中 e、t、o、a、n、i 等字母的出现频率，如果符合自然语言特征，则说明密文是用换位密码写的。第二步是猜测密钥的长度，即列数。在许多情况下，破译者根据消息的上下文，常常可以猜测出消息中可能包含的单词或短语，选择的单词或短语最好比较长一些，使其至少可能跨越两行，如"latestscheme"。将选择的单词或短语按照假定的长度 k 截成几行，由于同一列上相邻的字母在密文中必是相邻的，因此可以将各列上的各种字母组合记下来，在密文中搜索。比如将"latestscheme"按照假设的长度 8 截成两行，则相邻的字母组合有 lh、ae、tm 和 ee。假如设想的 k 是正确的，则大部分设想的字母组合在密文中都会出现；如果搜索不到，则换一个 k 再试。通过寻找各种可能性，破译者常常能够确定密钥的长度。第三步是确定各列的顺序。如果列数比较少，可以逐个检查 $k(k-1)$ 个列对，查看它们的二字母组的频率是否符合英文统计特征，与特征符合最好的列对认为其位置正确。然后从剩下的列中寻找这两列的后继列，如果某列和这两列组合后，二字母组和三字母组的频率都很好地符合英文统计特征，那么该列就是正确的后继列。通过同构法也可以找到它们的前趋列，直至最终将所有的列序全部找到。

(4) 位级加密。

位级加密(bit-level Encryption)是对构成这些字符的二进制位进行加密，密钥也是一个二进制位，一般是 64 位或者 128 位。明文先被划分成与密钥相同长度的二进制位串，然后再与密钥进行异或运算，运算结果就是密文。

【例 4-13】 如果明文是 1010111001100010，密钥是 1110010110000101，采用位级加密后的密文是什么？如何解密？

明文 1010111001100010　　　　密文 0100101111100111
密钥 1110010110000101　　　　密钥 1110010110000101
密文 0100101111100111　　　　明文 1010111001100010

由于异或操作是可逆的，对密文使用密钥再执行一次异或操作，就可以得到明文，加密和解密的密钥是相同的。密钥越长，位级加密就越安全，但仍然存在如何安全地将密钥传送给可信任的接收者的问题。

2) 密秘密钥算法

现代密码学也使用替代密码和换位密码的思想，但和传统密码学的侧重点不同。传统密码学的加密算法比较简单，主要通过加长密钥长度来提高保密程度；而现代密码学正

好相反，它使用极为复杂的加密算法，即使破译者能够对任意数量的选择明文进行加密，也无法找出破译密文的方法。

一种著名的加密技术是DES(Data Encryption Standard)，它是由IBM公司在20世纪70年代开发的。DES用56位的密钥，将64位的明文块转换为64位的密文，它有2^{56}种可能性。该算法有许多步骤，组合了替代和换位技术，每一步的输出是下一步的输入，最后一步产生加密后的64位密文。在接收端，要使用同一密钥执行相反的步骤进行解密。尽管DES算法很复杂，但它已被集成在VLSI芯片中，加密过程很快。近年来由于计算能力的迅速提高，一台高性能计算机可以在3到4小时内破解这种密码，因此又开发了三重DES(triple DES)，三重DES将密钥增加到112位，明文块首先用密钥的前56位加密，然后再用后56位加密，最后再用前56位加密，这样得到的密文需要尝试2^{112}次才能破译。

图4-27(a)是一个实现换位密码的基本部件，称为P盒(P-box)，它将输入顺序映射到某个输出顺序上，只要适当改变盒内的连线，它就可以实现任意的排列。图4-27(b)是一个实现替代密码的基本部件，称为S盒(S-box)，它由一个3-8译码器、一个P盒和一个8-3编码器组成。一个3比特的输入将选择3-8译码器的一根输出线，该线经P盒换位后从另一根线上输出，再经8-3编码器转换成一个新的3比特序列。将P盒和S盒相复合构成乘积密码系统，就可以实现非常复杂的加密算法。图4-27(c)是一个乘积密码系统的例子。一个12比特的明文经第一个P盒排列后，按3比特一组分成4组，分别进入4个不同的S盒进行替代，替代后的输出又经第二个P盒排列，然后再进入4个S盒进行替代，这个过程重复进行直至最后输出密文。只要级联的级数足够大，算法就可以设计得非常复杂。

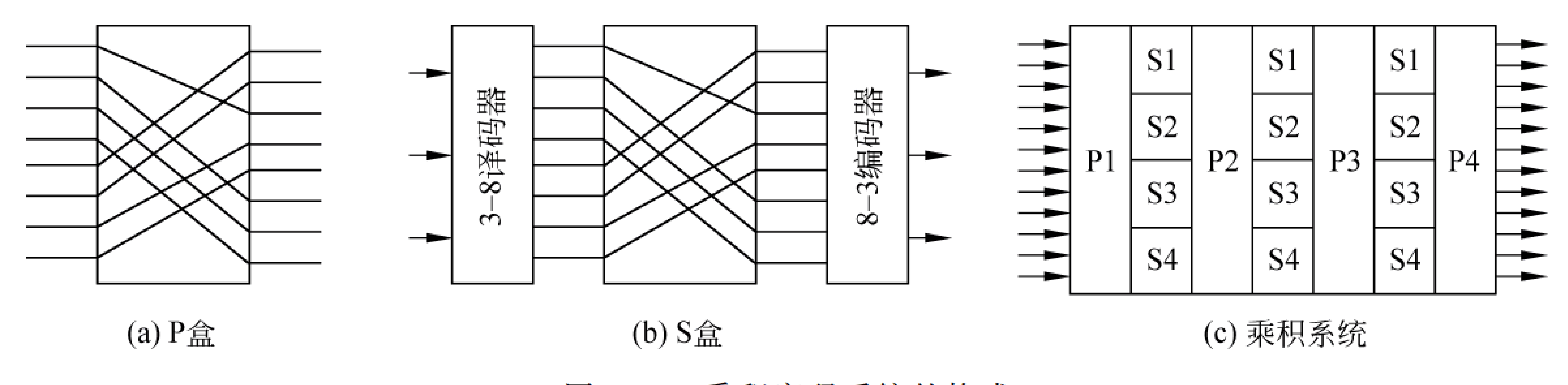


图4-27　乘积密码系统的构成

解密密钥必须和加密密钥相同，这就产生了如何安全地分发密钥的问题。传统上是由一个中心密钥生成设备产生一个相同的密钥对，并由人工信使将其传送到各自的目的地。对于一个拥有许多部门的组织来说，这种分发方式是不能令人满意的，尤其是出于安全方面的考虑需要经常更换密钥时更是如此。另外，两个完全陌生的人要想秘密地进行通信，就必须通过实际会面来商定密钥，否则别无他法。1976年，Diffie和Hellman提出了一种全新的加密思想——公开密钥算法，很好地解决了这个问题。

3）公开密钥算法

在公开密钥加密(Public Key Encryption，PKE)提出之前，所有密码系统的解密密钥和加密密钥都有直接的联系，即从加密密钥可以很容易地导出解密密钥，因此所有的密码

学家理所当然地认为应对加密密钥进行保密，但是 Diffie 和 Hellman 提出了一种完全不同的设想，从根本上改变了人们研究密码系统的方式。在 Diffie 和 Hellman 提出的方法中，使用非对称密钥(asymmetric key)，即加密密钥和解密密钥是不同的，并且从加密密钥不能得到解密密钥。

为此，加密算法 E 和解密算法 D 必须满足以下三个条件：$D(E(P))=P$，即将解密算法 D 作用于密文 $E(P)$ 后就可获得明文 P；从 E 导出 D 非常困难，即不可能从 E 导出 D；使用“选择明文”攻击不能攻破 E，即破译者即使能加密任意数量的选择明文，也无法破译密码。如果能够满足以上 3 个条件，则加密算法完全可以公开。

Diffie 和 Hellman 算法的基本思想是：如果某个用户希望接收秘密报文，他必须设计两个算法：加密算法 E 和解密算法 D，然后将加密算法放于任何一个公开的文件中广而告知，这也是公开密钥算法名称的由来，他甚至也可以公开他的解密方法，只要他妥善保存解密密钥即可。在这种算法中，每个用户都使用两个密钥，其中加密密钥是供其他人向他发送报文用的，这是公开的，解密密钥是用于对收到的密文进行解密的，这是保密的。公开密钥算法中，通常公开密钥(public key)和私人密钥(private key)分别称为加密密钥和解密密钥，以与传统密码学中的秘密密钥相区分。由于私人密钥只由用户自己掌握，不需要分发给别人，也就不用担心在传输的过程中泄密或被其他用户泄密，因而是极其安全的。用公开密钥算法解决上面所说的密钥分发问题非常简单，中心密钥生成设备产生一个密钥后，用各个用户公开的加密算法对之进行加密，然后分发给各用户，各用户再用自己的私人密钥进行解密，既安全又省事。两个完全陌生的用户之间也可以使用这种方法很方便地商定一个秘密的会话密钥。

例如，当用户 A 和 B 希望秘密通信时，A 和 B 各自可以从公开的文件中查到对方的加密算法。若 A 需要将秘密报文发给 B，则 A 用 B 的加密算法 EB 对报文进行加密，然后将密文发给 B，B 使用解密算法 DB 进行解密，而除 B 以外的任何人都无法读懂这个报文；当 B 需要向 A 发送消息时，B 使用 A 的加密算法 EA 对报文进行加密，然后发给 A，A 利用 DA 进行解密。

由于公开密钥算法潜在的优越性，研究者们一直在努力寻找符合以上 3 个条件的算法，已经有一些算法被提了出来，其中较好的一个是由 MIT 的一个研究小组提出的，并以 3 位发现者名字的首字母进行命名，称为 RSA 算法。RSA 算法使用的解密密钥通常是两个很大的素数，加密密钥是由这两个素数产生的。理论上讲，如果这两个素数足够大，就几乎不可能从它们的乘积(加密密钥)反推出原来的两个素数(解密密钥)。在实际应用中，素数一般选得很大，在 10^{130} 到 10^{310} 之间。在此不对它做理论上的推导，只说明如何使用这种算法。

(1) 选择两个大素数 p 和 q(典型值为大于 10^{100})。

(2) 计算 $n=p\times q$ 和 $z=(p-1)\times(q-1)$。

(3) 选择一个与 z 互质的数，令其为 d。

(4) 找到一个 e 使满足 $e\times d=1(\mathrm{mod}\ z)$。

计算以上参数后，就可以开始对明文加密。首先将明文看成是一个比特串，将其划分成一个个的数据块 P 且有 $0\leqslant P<n$。要做到这一点并不难，只需先求出满足 $2^k<n$ 的最

大 k 值，然后使得每个数据块长度不超过 k 即可。对数据块 P 进行加密，计算 $C=P^e(\mathrm{mod}\ n)$，C 即为 P 的密文；对 C 进行解密，计算 $P=C^d(\mathrm{mod}\ n)$。可以证明，对于指定范围内的所有 P，其加密函数和解密函数互为反函数。进行加密需要参数 e 和 n，进行解密需要参数 d 和 n，所以公开密钥由 (e,n) 组成，私人密钥由 (d,n) 组成。

【例 4-14】 假设取 $p=3$，$q=11$，则公开密钥和私人密钥分别是什么？若要加密的明文为 $P=4$，则对应密文是什么？接收方如何解密？

由 p 和 q，计算出 $n=33$ 和 $z=20$。由于 7 和 20 没有公因子，因此可取 $d=7$；解方程 $7e=1(\mathrm{mod}\ 20)$ 可以得到 $e=3$。由此公开密钥为(3,33)，私人密钥为(7,33)。再由 $C=P^e(\mathrm{mod}\ n)=4^3(\mathrm{mod}\ 33)=31$，计算出对应的密文为 $C=31$。接收方收到密文后进行解密，计算 $P=C^d(\mathrm{mod}\ n)=31^7(\mathrm{mod}\ 33)=4$，恢复出原文。

RSA 算法的安全性建立在难以对大数提取因子的基础上，如果破译者能对已知的 n 提取出因子 p 和 q 就能求出 z，知道了 z 和 e，就能利用 Euclid 算法求出 d。所幸的是，三百多年来虽然数学家们已对大数因式分解的问题作了大量研究，但并没有取得什么进展，到目前为止这仍是一个极其困难的问题。据 Rivest 等人的推算，用最好的算法和指令时间为 $1\mu s$ 的计算机对一个 200 位的十进制数作因式分解需要 40 亿年的机器时间，而对一个 500 位的数作因式分解需要 1025 年。即使计算机的速度每 10 年提高一个数量级，能做 500 位数的因式分解也是在若干世纪之后，然而到那时，人们只要选取更大的 p 和 q 就行了。

非对称密钥技术的最大优点是解决了密钥交换的问题，所以广泛应用于许多产品中，如 Netscape Navigator、Lotus Notes 和 Internet Explorer，用于因特网上对数据的加密传输。另一个在电子邮件、计算机数据和语言通信中使用的非对称加密/解密程序是 PGP (Pretty Good Privacy)，它在 1991 年由 Phillip R. Zimmerman 开发并免费发布到因特网上的，是第一个能使人们方便加密报文的产品。

微软公司在操作系统的 NTFS 中使用了 Encrypting File System，它是 PKE(Public Key Encryption)的一种。用户选择文件后右击，选择“属性”命令，然后在“常规”选项卡中单击“高级”按钮，再选择“加密内容以便保护数据”复选框，即可加密文件或文件夹。此外还有许多加密产品，如 WinZip 9.0 可对压缩文件进行 128 位和 256 位 AES 加密，ScramDisk 可提供 64 位和 128 位加密，DriveCrypt 可提供 1344 位军方标准的加密。

2. 数字签名和消息认证

前面所介绍的对称密钥体制和公开密钥体制都是围绕着如何确保信息的保密性而展开的，其主要思想是通过不同的加密策略和算法将明文转变为密文后再传输。除了信息的保密之外，如何保证信息的来源方是真实的，保证收到的信息是可靠的而没有被非法篡改，也是非常重要的，这就是认证。认证包括对用户身份的认证和对消息正确性的认证两种方式。用户认证用于鉴别用户的身份是否是合法用户，可以利用数字签名技术来实现的；而消息认证主要用于验证所收到的消息确实是来自真正的发送方且未被修改的消息，也可以验证消息的顺序和及时性。

1）数字签名

目前，数字签名技术已经广泛应用于商业、金融、军事等领域，特别是电子邮件、电子资金转账（Electronic Funds Transfer，EFT）、电子数据交换（Electronic Data Interchange，EDI）、软件分发数据存储和数据完整性检验的应用。数字签名是以电子形式存在于数据信息之中的数字化“签名”，可用于辨别签署人的身份，并表明签署人对数据信息中包含的信息的认可。为了使数字签名能代替传统的签名，必须保证能够实现以下功能：接收者能够核实发送者对消息的签名；签名具有不可否认性；接收者无法伪造对消息的签名。

这里用一个例子来说明上述 3 个功能的用途。假设 A 和 B 分别代表一个股民和他的股票经纪人。A 委托 B 代为炒股，并指令当他所持的股票达到某个价位时立即全部抛出。B 首先必须认证该指令确实是由 A 发出的，而不是其他什么人在伪造指令，这就需要第一个功能。假定股票刚一卖出，股价立即猛升，A 后悔不已。如果 A 是不诚实的，他可能会控告 B，宣称他从未发出过任何卖出股票的指令。这时 B 可以拿出有 A 亲自签名的委托书作为最有力的证据，这就需要第二个功能。另一种可能是 B 玩忽职守，当股票价位合适时没有立即抛出，不料此后股价一路下跌，客户损失惨重。为了推卸责任，B 可能试图修改委托书中关于股票临界价位为某一个实际上不可能达到的值。为了保障客户的权益，于是需要第三个功能。

2）消息认证

在消息认证中，最常用的是消息认证码（MAC）和散列函数。消息认证码（或称密码校验和）是在一个密钥的控制下将任意长的消息映射到一个简短的定长数据分组，并将它附加在消息后。接收者通过重新计算 MAC 来对消息进行认证，如果收到的 MAC 与计算得出的 MAC 相同，则接收者可以认为消息未被篡改过；否则认为消息被篡改过。消息认证码对于要保护的信息来说是唯一的且与消息一一对应，因此可以在一定程度上有效地保护消息的完整性。

散列函数（又称哈希函数，杂凑函数）是对不定长的输入产生定长输出的一种特殊函数，记为 $H(M)$ 。其中 M 是变长的消息，$H(M)$ 是定长的散列值，称为密码散列值、密码校验和、密码指纹或消息摘要。由于散列函数本身公开，传送过程中对散列值需要另外的加密保护，如果没有对散列值的保护，篡改者可以在修改消息的同时修改散列值，从而使散列值的认证功能失效。MD5 曾经是使用最普遍的安全散列算法，但自从 2004 年 9 月国际密码年会 MD5 算法被破解以后，消息认证码的传统实现途径也将改变，寻找一种足够安全的单向散列算法已经成为当务之急。

数字签名总是和报文摘要结合起来使用的，如图 4-28 所示。发送报文时，发送方用一个散列函数从报文中生成报文摘要，然后用自己的私人密钥对这个摘要进行加密，这个加密后的摘要将作为报文的数字签名和报文以及公开密钥一起发送给接收方。接收方首先用与发送方一样的散列函数从接收到的原始报文中计算出报文摘要，接着再用接收到的公开密钥来对报文附加的数字签名进行解密，如果这两个摘要相同，那么接收方就能确认该数字签名是发送方的。

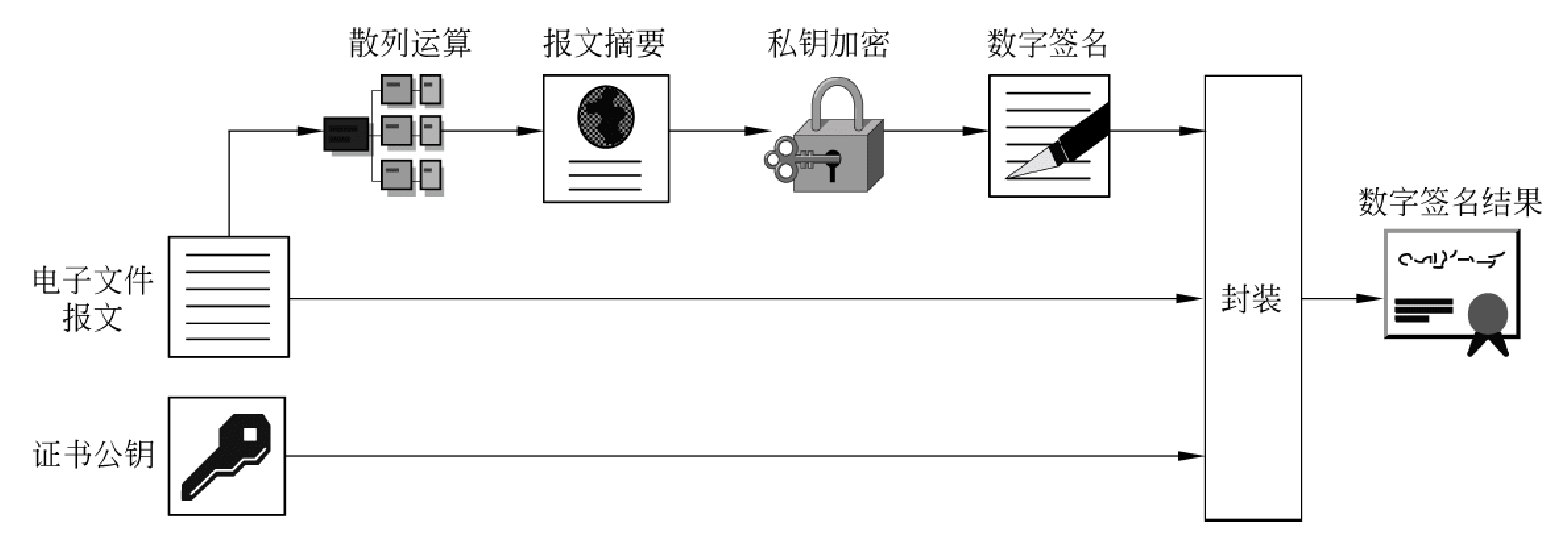


图 4-28　数字签名生成的过程

3. 数字证书

就像对称密钥密码学一样，密钥管理和分发也是公钥密码面临的问题。除了保密性之外，公钥密码学的一个重要的问题就是公钥的真实性和所有权问题。为此，人们提出了一种很好的解决办法——公钥证书(Public Key Certificate，PKC)，又称为数字证书，公钥证书可以证实一个公钥与某一用户身份之间的绑定。为了提供这种绑定关系，需要一个可信第三方实体来担保用户的身份，该第三方实体称为认证机构(简称 CA，即 Certification Authority)，它向用户颁发证书，证书中包含证书所有人的名称、证书所有人的公开密钥、证书发行者对证书的签名、证书的序列号(每个证书都有一个唯一的证书序列号)以及证书公开密钥的有效日期等内容。

数字证书提供了一种在因特网上验证身份的方式，其作用类似于司机的驾驶执照或日常生活中的身份证。它是进行安全通信的必备工具，能够保证信息传输的保密性、数据完整性、不可抵赖性以及交易者身份的确定性。当在网上银行进行金融交易时，为了保证交易的安全，往往需要一个称为 U 盾的 USB 设备，它的外形类似于一个 U 盘，在进行网上银行操作时需要插在计算机的 USB 接口上，这个 U 盾中存储的内容其实就是数字证书。

在图 4-28 中，若电子文件报文是包含公开密钥持有者信息的文件，那么数字签名的结果就是数字证书。只不过，用于加密数字摘要的私有密钥是由 CA 所拥有的。

公钥基础设施(Public Key Infrastructure，PKI)采用证书管理公钥，在功能上主要由以下几个部分组成：

- CA(Certification Authority)：签发证书和证书撤销列表，对证书和密钥进行管理。
- RA(Registration Authority)：对用户身份的真实性进行核对，处理用户的注册请求。
- 证书持有者：CA 为其签发证书。证书持有者可以用证书进行数字签名或加密。
- 依赖证书的实体/证书使用者(certificate-relying party/certificate user)：通过可信 CA 的公钥验证对 PKC 的签名和 PKC 的可信路径。
- 仓库(repository)：存储和发布证书、证书撤销列表等信息。

4.4.3 网络安全技术

几乎所有上网用户都遭受过计算机病毒和木马的袭扰，对于一些机构的计算机网络，还会受到入侵攻击。本节将对网络安全技术进行简介，并使读者了解如何检测、避免或减轻这些安全隐患的威胁。

1. 网络防火墙

防火墙是指由软件和硬件设备组合而成的，在局域网和因特网之间建立的一个安全网关(security gateway)，能够保护内部网免受非法用户的侵入(图 4-29)。在逻辑上，防火墙是一个分离器，也是一个分析器，有效地监控了内部网和因特网之间的任何活动，保证了内部网络的安全。

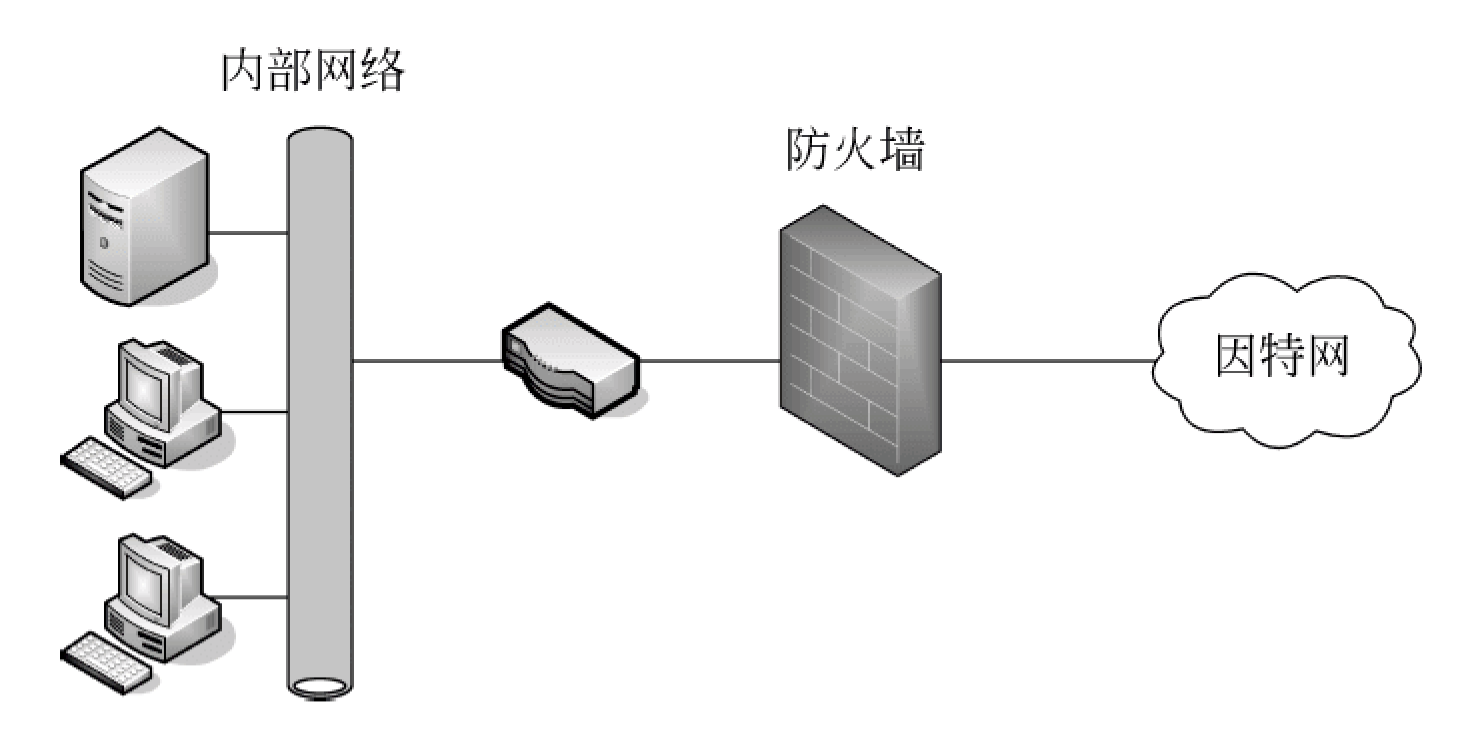


图 4-29 防火墙位于内网与因特网之间

防火墙按照所采用的技术可以分为包(分组)过滤型、应用代理型和状态监视型三大类。无论一个防火墙的实现过程多么复杂，归根结底都是在这三种技术的基础上进行功能扩展的。

1) 包过滤型防火墙

包过滤是防火墙最基本的实现形式，它可以根据访问规则控制哪些数据包可以进出网络，而哪些数据包不允许进出网络。适当的过滤规则可以让防火墙工作得更安全有效，但是这种技术只能根据预设的过滤规则进行判断，一旦出现一个落在规则之外的有害数据包请求，整个防火墙的保护就会失效。

为了避免上述情况发生，人们对包过滤技术进行了改进，这种改进后的技术称为动态包过滤，动态包过滤功能在保持了原有静态包过滤技术和过滤规则的基础上，会对已经成功与计算机连接的报文传输进行跟踪，并且判断该连接发送的数据包是否会对系统构成威胁，一旦触发其判断机制，防火墙就会自动产生新的临时过滤规则或者对已经存在的过滤规则进行修改，从而阻止该有害数据的继续传输。但是由于动态包过滤需要消耗额外的资源和时间来提取数据包内容进行判断处理，所以与静态包过滤相比，它会降低运行效率。

基于包过滤技术的防火墙，其缺点是很显著的，它得以正常工作的一切依据都在于过滤规则的实施，但又难以建立满足各种需求的精细规则表，因为规则数量和防火墙性能成

反比。另外，它只能工作于网络层和传输层，不能判断高级协议中的数据是否有害。

2）应用代理型防火墙

这种防火墙实际上就是一台小型的带有数据检测过滤功能的透明代理服务器(transparent proxy)，但是它并不是单纯地在一个代理设备中嵌入包过滤技术，而是采用了一种被称为“应用协议分析”(application protocol analysis)的新技术。

应用协议分析技术工作在OSI/RM模型的最高层——应用层上，在这一层里能够接触到的所有数据都是最终形式（支持HTTP、HTTPS/SSL、SMTP、POP3、IMAP、NNTP、TELNET、FTP、IRC等常用应用协议），也就是说，防火墙“看到”的数据和最终用户看到的是一样的，而不是一个个带着地址端口协议等原始属性信息的数据包，因而它可以实现更高级的数据检测目标。代理防火墙把自身映射为一条透明线路，在用户和外界看来，它们之间的连接并没有任何阻碍，但是这个连接的数据收发实际上是经过了代理防火墙的中转，当外界数据进入代理防火墙的客户端时，“应用协议分析”模块便根据应用层协议处理这个数据，通过预置的处理规则查询这个数据是否带有危害，由于这一层面对的已经不再是组合有限的报文协议，甚至可以识别类似于“GET/sql.asp?id=1 and 1”的数据内容，所以防火墙不仅能根据应用层提供的信息判断数据，更能像管理员分析服务器日志那样观察内容，分辨危害。同时，由于工作在应用层，防火墙还可以实现双向限制，在过滤外部网络有害数据的同时也监控内部网络的信息，管理员可以配置防火墙实现身份验证和连接时限的功能，进一步防止内部网络信息泄露的隐患。最后，由于代理防火墙采取代理机制进行工作，内外部网络之间的通信都需先经过代理服务器审核，通过后再由代理服务器连接，根本没有给分隔在内外部网络两边的计算机直接会话的机会，可以避免入侵者使用“数据驱动”攻击方式渗透内部网络，可以说，应用代理技术是比包过滤技术更完善的防火墙技术。

3）状态监视型防火墙

这是继包过滤技术和应用代理技术之后出现的防火墙技术，状态监视(stateful inspection)技术在保留了对每个数据包的头部、协议、地址、端口、类型等信息进行分析的基础上，进一步发展了会话过滤(session filtering)功能，在每个连接建立时，防火墙会为这个连接构造一个会话状态，里面包含了这个连接数据包的所有信息，以后这个连接都基于这个状态信息进行，这种检测的高明之处是能对每个数据包的内容进行监视，一旦建立了一个会话状态，则此后的数据传输都要以此会话状态作为依据。例如，一个连接的数据包源端口是8000，那么在以后的数据传输过程中防火墙都会审核这个包的源端口还是不是8000，如果不是，这个数据包就被拦截。而且会话状态的保留是有时间限制的，在超时的范围内如果没有再进行数据传输，这个会话状态就会被丢弃。状态监视可以对包内容进行分析，从而摆脱了传统防火墙仅局限于几个包头部信息的检测弱点，而且这种防火墙不必开放过多端口，进一步杜绝了可能因为开放端口过多而带来的安全隐患。

由于状态监视技术相当于结合了包过滤技术和应用代理技术，因此是最先进的，但是由于其实现技术复杂，在实际应用中还不能做到真正完全有效的数据安全检测，而且在一般的计算机硬件系统上很难设计出基于此技术的完善防御措施。

2. 入侵检测

随着网络的不断扩大,网络环境变得越来越复杂,网络攻击方式不断翻新,对于网络安全来说,单纯的防火墙技术暴露出明显的不足和弱点:许多攻击(如 DoS 攻击,会伪装成合法的数据流)可以绕过通常的防火墙;防火墙不具备实时入侵检测能力及对于病毒束手无策等。在这种情况下,网络的入侵检测系统(Intrusion Detection System,IDS)在网络的整个安全系统解决方案中就显示出极大的作用。它可以弥补防火墙的不足,为网络安全提供实时的入侵检测及相应的防护手段。一个合格的入侵检测系统能大大简化管理员的工作,保证网络安全地运行。入侵检测系统是对防火墙的必要补充,作为重要的网络安全工具,它可以对系统或网络资源进行实时检测,及时发现闯入系统或网络的入侵者,也可预防合法用户的误操作。它对计算机和网络资源的恶意使用行为进行识别和响应,它不仅检测来自外部的入侵检测行为,同时也监督内部用户的未授权行为。

1) 入侵检测系统

入侵是指试图破坏计算机机密性、完整性、可用性或可控性的活动集合。入侵行为包括非授权用户试图存取数据,处理数据,或者妨碍计算机的正常运行。入侵检测就是对企图入侵、正在进行的入侵或已经发生的入侵进行识别的过程。而从网络或计算机系统中收集信息进行入侵检测的系统被称为入侵检测系统。入侵检测系统主要包括 3 个功能部件:信息收集、信息分析和结果处理,如图 4-30 所示。其基本工作原理是:从不同环节收集信息并分析该信息,试图寻找入侵活动的特征,并对自动检测到的行为做出响应,记录并报告检测过程和结果。



图 4-30　入侵检测系统基本框架

2) 入侵检测系统分类

根据数据源的不同,入侵检测系统主要分成 3 种:

(1) 基于主机的入侵检测系统(HIDS)保护的目标是主机,它往往通过监视主机端的用户行为、系统日志、应用程序日志及系统调用来收集信息。主机入侵检测的优点在于信息来源准确广泛,能够充分利用操作系统提供的功能,容易定义正常行为,结合异常分析可以有较好的检测效果。缺点在于需要为不同操作系统平台分别开发程序,会造成所在主机性能下降、安装数量众多等问题。

(2) 基于网络的入侵检测系统(NIDS)保护的是其所在的整个网段,它的数据源就是网络上的数据包。通过对截获数据的分析来发现是否有恶意入侵行为发生。它的优点是独立于平台,不会给网络造成额外负担,容易部署等。但是由于 NIDS 被动监听的工作原理使得入侵者有可能绕过或欺骗检测系统,同时使 NIDS 在入侵响应方面也处于被动的状态。另外,对加密数据的分析仍然是 NIDS 的一个薄弱环节。

(3) 混合入侵检测系统综合了上述两类入侵检测系统的特点,既监视主机数据也监视网络数据,它通常是基于分布式部署,有一部分传感器(或称为代理)驻留在主机上收集

信息，另一部分则部署在网络中，它们都由中央控制台管理，将收集到的信息发往控制台进行处理。

根据检测原理的不同，入侵检测系统可以分为两种：

(1) 基于异常的入侵检测(anomaly detection)技术有一个假设，就是入侵和滥用行为不同于一般正常用户或者系统的行为。异常检测先在用户、系统或者网络正常操作的一段时间收集事件和行为的信息，再根据这些信息建立正常或者有效行为的模式。在检测的时候，通过某种度量，计算事件的行为偏离正常行为的程度。把当前行为和正常模式比较，如果偏离程度超过一定的范围，则报警异常。异常检测的本质就是查找一些被认为是异常的行为。换句话说，所有不符合正常模式的行为都被认为是入侵。“模式”通常使用一组系统的度量来定义。所谓“度量”，则是系统/用户行为在特定方面的衡量标准。每个度量都对应一个限值或相关的变动范围。因此，异常检测系统经常会有一些误警行为。

(2) 基于误用的入侵检测(misuse detection)模型的特点是收集入侵行为的特征，建立相关的特征库，在后续的检测过程中，将收集到的数据与特征库中的特征代码进行比较(即模式匹配)，得出是否入侵的结论。可以看出，这种模型与主流的病毒检测方式基本一致。当前流行的入侵检测系统基本上采用的是这种模型。其优点是误报少，缺点是只能发现攻击库中已知的攻击，且其复杂性将随着攻击数量的增加而增加。

习　　题

1. 以下所列出的设备，哪些是计算机网络的功能设备？

A. 笔记本电脑　B. 鼠标/键盘　C. 电话机　D. 服务器

E. 双绞线　F. USB 数据线　G. 通信卫星　H. 光纤

I. 网络交换机　J. 路由器　K. 多路复用器　L. U 盘

M. 网络接口卡　N. 扫描仪　O. 显示器　P. 打印机

Q. 多媒体音箱　R. 调制解调器　S. 智能手机　T. 平板电脑

2. 现代计算机网络为什么要将报文分割成一个个分组来进行传输？说出你认为最重要的一个理由。

3. 计算机网络按照其规模大小和延伸距离远近划分为(　　)、(　　)和(　　)。

4. 计算机通过点到点的链路与中心结点相连，具有这种拓扑结构的网络称为(　　)。

A. 因特网　B. 星形网　C. 环形网　D. 总线型网

E. 广域网

5. 计算机网络中常用的有线传输介质有(　　)。

A. 双绞线、红外线、同轴电缆　B. 同轴电缆、激光、光纤

C. 双绞线、同轴电缆、光纤　D. 微波、双绞线、同轴电缆

6. 采用全互连拓扑结构建设一个具有 500 个结点的广域网。假定网络中结点之间的平均距离为 50 千米，每千米的线路建设成本是 1 万元。建设此广域网的线路总成本是多少？通过这个结果你能得到什么结论？

7. 如果你在家里构建了一个能够连接因特网的家庭局域网。请构思一个能够利用家庭局域网为你的家居生活带来方便的应用。简要说明你的构思大致的实现方法。
8. 哪一种网络应用模式对客户端的要求最低？哪一种网络应用模式对所有主机的要求是等同的？
9. 因特网的体系结构是怎样的？画出因特网体系结构的层次图，并在图上标注每层的名字。
10. 说出一个生活中使用分层思想的案例，并对其中的分层机制进行简要的叙述。
11. 网络协议的关键要素包括语法、(　　)和(　　)。
12. 中国一家公司的经理要与德国一家公司的经理进行商务谈判。请将谈判过程的机制用层次结构表示，给每个层次用一个贴切的词汇命名，说明每个层次的功能和对等层之间的协议是什么，最后详细描述在这种机制下的商谈过程。已知条件如下：
    (1) 谈判策略已经由双方的董事会各自确定，由双方的经理亲自掌握。
    (2) 中方经理不懂德语，德方经理也不懂汉语，但双方都可以聘请翻译人员。
    (3) 翻译人员只负责语言翻译，不涉及商务。
    (4) 双方的通信手段只能使用传真，而且只有秘书会使用传真。
13. 一个具有5层体系结构的网络，其每一层添加的报文头部长度为20B。假定发送方要传输一份500B的报文给接收方(不考虑报文分段)，物理介质上传输的总位数是多少？网络的传输效率是多少？接收方收到的报文长度是多少？
14. TCP/IP参考模型共分为4层，分别是(　　)、(　　)、(　　)和(　　)。
15. 因特网上的每一台主机都有一个唯一的、可识别的主机地址，称为(　　)。
    A. 端口号　　B. 物理地址　　C. IP地址　　D. 域名
    E. MAC地址
16. IP地址是一个(　　)位的二进制数。
17. 以下IP地址中(　　)和(　　)属于同一子网(子网掩码为255.255.192.0)。
    A. 150.20.115.133　　B. 150.20.190.2
    C. 150.20.192.59　　D. 150.20.215.133
18. 用户主机上打开了两个IE浏览器窗口，浏览同一个网站的不同网页。该网站的Web服务器如何知道将网页发送到哪个IE浏览器窗口？如果同时浏览不同网站的主页，各网站对应的Web服务器如何知道将网页发送到哪个IE浏览器窗口？
19. DNS系统用于以下(　　)任务。
    A. 将IP地址转换为MAC地址　　B. 将域名转换为MAC地址
    C. 将域名转换为IP地址　　D. 将IP地址转换为MAC地址
20. 一般情况下，通过域名访问一个网站需要访问几次根域名服务器就能查找到该域名服务器的IP地址？
21. 因特网中的地址有域名、IP地址和MAC地址之分，为什么需要这么多不同类型的地址？只要其中一种地址行不行？试说明理由。
22. 局域网设置网关的作用是什么？
23. 什么情况下，网关才会将收到的IP分组发送到因特网上？

24. 万维网 WWW 的 3 个组成部分是(　　)、(　　)和(　　)。

25. 在发送邮件过程中要建立几个 TCP 连接?

26. 以下说法中(　　)是正确的。

A. 万维网是一种广域网

B. 万维网就是因特网

C. 因特网是一种基于报文交换的网络

D. 因特网是一种路由器网络

27. 在一次 FTP 传输中要建立几个 TCP 连接? 每个 TCP 连接的作用分别是什么?

28. 下列攻击中,(　　)属于主动攻击。

A. 无线截获　　B. 搭线监听　　C. 拒绝服务　　D. 流量分析

29. 数据加密技术一般有两种类型,分别是(　　)加密和(　　)加密。

30. 假设密钥 $k=8$,用替代密码将明文 XIANJIAOTONGUNIVERSITY 加密。

31. 在网络购物过程中,用到了哪些安全技术? 它们分别用于网络购物的哪个步骤?

32. 在网上传输的数字证书中包括报文明文、证书公钥和数字签名。但证书公钥并没有被加密,而是直接封装在证书中。不加密的原因是什么? 如果对公钥加密再封装到证书中会出现什么问题?

33. 防火墙有哪些种类? 哪一种防火墙的安全性更好一些? 为什么?



第5章 C程序设计基础

引言

计算机程序是一系列的指令，它能使计算机执行特定的任务。编程语言，如 Visual C++ 用于将人们能懂得的指令翻译成计算机可以理解和执行的步骤。深入计算机的最底层，作为计算机"心脏"的微处理器，只能理解数字指令（机器指令）。这种处理器所能理解的指令是极简单的命令，这些命令大多数只能在存储地址之间移动数据。这些处理器懂得的命令称为机器语言，即 PC 可使用的最基本的语言。

机器语言称为低级语言，因为它是处理器能够懂得的最低级的方式。用机器语言编写程序是一项极其烦琐的任务。幸运的是，用户不必用它来编写计算机程序。有更高级的编程语言已经开发出来，使一般用户能够编写程序。这些高级编程语言允许程序员以类似英文的方式编写指令，然后将指令转化成处理器能懂的含有机器语言指令的程序。

教学目的

- 掌握 C 程序的基本结构。
- 掌握变量、表达式及运算符的基本用法。
- 熟悉编写 C 程序的方法。
- 掌握 C 的流程控制语句。

5.1 程序设计基础

5.1.1 什么是程序设计

计算机之所以比电视机、DVD 机、计算器等其他电子设备功能更灵活，是因为计算机软件的"可编程"，也就是说，同样的硬件配置，加载不同的软件就可以完成不同的工作。

当用户使用计算机来完成某项工作时，会面临两种情况：一种是可以借助现成的应用软件完成，如文字处理可使用 Word，表格处理可使用 Excel，科学计算可选择 Matlab，

绘制图形可使用 Photoshop 等;另一种情况是,没有完全合适的软件可供使用,这时就需要使用计算机语言来编制完成某个特定功能的程序,这就是程序设计(programming)。

考虑这样一个任务:统计大学中一个班学生的考试成绩,并选出优秀学生。

可以看出,这是一个很简单的任务。如果由人去做,是一件比较轻松的事,只需要几张纸和一支笔,一个小时就完成了。对这样简单的任务,可以不需要计算机去做。但如果将任务修改为:要求对一个大学中的 4 万名学生的英语成绩进行管理,分别统计出 100 至 60 各个分数段的人数,并对过去 5 年中的数据进行对比分析。这个任务由人工完成会比较麻烦,完全值得设计计算机程序。

功能完善的商业程序一般都是比较大的,一个字处理软件可能包含 75 万行左右的代码,而按照美国国防部的标准,少于 10 万行代码称为小程序,超过 100 万行才是大程序。为便于理解,下面还是以微小的程序作为例子来介绍程序设计的概念。

计算机程序设计是用计算机语言编写一些代码(指令)来驱动计算机完成特定的功能,是问题求解过程的关键步骤之一。我们已经知道,对复杂的系统性问题的求解,需要经过需求分析、软件设计、程序编制和调试、系统测试、文档编写等一系列活动,甚至还包括硬件系统配置、人员配置、开发方法及开发工具选择等相关的各种任务管理。而对于小型问题(或称算法类问题),这个过程可以简化为问题描述、算法设计、代码编制以及调试运行。

5.1.2 程序设计语言

在过去的几十年里,人们根据描述问题的需要而设计了数千种专用和通用的计算机语言,有的语言是为了编写系统软件而重在提高效率(如 C 语言);有些是为了提高程序设计速度,面向商业应用(如 COBOL 和 dBASE);还有些语言是为了用于教学(BASIC 和 Pascal 都属于此类)。这些语言中只有少部分得到了比较广泛的应用。

1. 计算机语言的分类

对程序设计语言的分类可以从不同的角度进行,如面向机器的程序设计语言、面向对象的程序设计语言、面向过程的程序设计语言等。其中,最常见的分类方法是根据程序设计语言与计算机硬件的联系程度,可以将其分为机器语言、汇编语言和高级语言 3 种类型。前两种类型的语言紧密依赖于计算机硬件,有时也统称为低级语言;而高级语言与计算机硬件关系较小。可以说,程序设计语言的演变经历了由低级向高级的发展过程。

1) 机器语言

机器语言(machine language)是直接用机器指令的集合作为程序设计手段的语言,而机器指令是以计算机所能理解和执行的以 0 和 1 组成的二进制编码表示的命令,它是所有语言中唯一能够被计算机直接理解和执行的指令。如第 2 章中所述,机器指令由操作码和操作数组成,其具体的表现形式和功能与计算机系统的结构相关联。

机器语言是面向机器的语言,其优点是计算机能够直接识别,执行效率高;缺点是因与具体机器相关,可移植性很差。且由于是由 0 和 1 这样的编码表示,在记忆、书写、编程、可读性等方面都比较困难。表 5-1 是分别用二进制和十六进制数表示的机器指令代

码,功能是求两数之和。由此可以看出,机器指令使用起来是很困难的。

表 5-1　求两数之和的机器语言指令

二进制表示	十六进制表示
1010 0000 0000 0000 0000 0000	A00000
0000 0011 0000 0110 1100 1000 0000 0000	0306C800
1010 0011 1100 1100 0000 0000	A3CC00

2）汇编语言

为了克服机器语言的缺点,人们采用了助记符与符号地址来代替机器指令中的操作码与操作数。如用 ADD 表示加法操作,用 SUB 表示减法操作,且操作数可用二进制、八进制、十进制和十六进制数表示。这种表示计算机指令的语言称为汇编语言(assembly language)。汇编语言也是一种面向机器的语言,但计算机不能直接执行汇编语言程序。用它编写的程序必须经过汇编程序翻译成机器指令后才能在计算机上执行。目前,由于它比机器语言可理解性好,比其他语言执行效率高,许多系统软件的核心部分仍采用汇编语言编制。

表 5-1 中的机器语言指令代码若用汇编语言编写,可表示为如下 3 行指令:

```
MOV  AX,  DATA1
ADD  AX,  DATA2
MOV  SUM, AX
```

3）高级语言

高级语言是更接近自然语言和数学语言的程序设计语言,是面向应用的计算机语言,与具体的计算机硬件平台无关。它的主要优点是符合人类叙述问题的习惯,而且简单易学。目前的大部分程序设计语言都属高级语言,其中使用较多的有 BASIC(Visual Basic)、Pascal(Delphi)、FORTRAN、COBOL、C、C++、Java 等。

用 Visual Basic 语言完成表 5-1 的程序功能只需要下面一行语句:

```
SUM=DATA1+DATA2;
```

可以看出,这很接近于自然语言,便于人理解。

目前高级语言正朝着非过程化发展,即只需告诉计算机“做什么”,而“怎样做”则由计算机自动处理。高级语言的发展将以更加方便用户使用为宗旨。

2. 程序语言的语法和语义

计算机语言有多种,不同的语言在语法和语义上都存在一定的差异。通俗地讲,语义是程序语言所表示功能的描述。比如,给一个数赋值,比较两个数大小,在屏幕上显示文字等。

不同的计算机语言可能都具有某些相同的功能定义,但在表现形式上却有不同,也就是语法不同。不同计算机语言的区别主要表现在语法(词法)上,即对同一种功能(相同语义),不同的语言可能有不同的表现形式。比如给一个变量赋值,下面的两种语言就有两种不同的表示方法:

```
sum:=eng+math          (Pascal 语言的赋值语句)
sum=eng+math;          (C 语言的赋值语句)
```

又如：要表示“如果数学成绩(math)和英语成绩(eng)都大于 80 就显示姓名(name)和分数合计(sum)”。C 语言的语句是

```
if(math>80 && eng>80)
    printf("第一名：%s %d",name,sum);
```

而如果用 Basic 语言表示，则是：

```
IF math>80 AND eng>80 THEN
    PRINT name1,sum1
END IF
```

可以看出，不同的语言用不同的符号表达了完全相同的含义。但很多时候，语言的差异还体现在语义上，即功能的描述上。比如在 C 和 Pascal 中的指针，在很多语言中就没有对等的体现。

3. 程序执行的起始点

程序都会从起始点开始执行，但不同的语言对起始点的处理方法有所不同。BASIC 语言中，程序从第一条语句开始执行，不论它是什么(这应当是最直观、最容易理解的方式了)；在 C 程序和 Visual Basic 控制台程序中，程序会从 main 函数的第一条语句开始执行，而不论 main 函数处于程序的什么位置(如果没有 main 函数，则无法运行)。

以下是一个 C 语言编写的在屏幕上显示“Hello World!”的程序段。

```
int main()
{
    printf("Hello World");        //程序从 main 函数的第一条语句开始执行
    return 0;
}
```

程序从起始点开始，按照程序员书写的顺序一条条执行指令。第一条语句先执行，接下来是第二条指令……一直到程序结尾。如果遇到一个子程序，则中断当前程序而转去执行子程序，执行完返回刚才的断点继续执行。

除了顺序执行外，程序执行还有分支、循环等多种控制。程序的控制结构将在 5.7 节中讨论。

5.1.3 程序的编译

在计算机语言中，用除机器语言之外的其他语言书写的程序都必须经过翻译，变成机器指令，才能在计算机上执行。因此，各种用于计算机程序设计的语言都必须配备相应的“翻译程序”。

编译是指将用高级语言编写好的程序(又称源程序、源代码)，经编译程序翻译，形成

可由计算机执行的机器指令程序(称为目标程序)的过程。如果使用编译型语言,必须把程序编译成可执行代码。因此编制程序需要三步:写程序、编译程序和运行程序。一旦发现程序有错,哪怕只是一个错误,也必须修改后再重新编译,然后才能运行。幸运的是,只要编译成功一次,其目标代码便可以反复运行,并且基本上不需要编译程序的支持就可以运行。

5.1.4 C 程序基本结构

首先来看一个简单例子,以便让读者对 C 语言编写的程序有一个初步的认识。

【例 5-1】 第一个 C 程序,在计算机屏幕上显示“Hello World!”。

程序:

```
//屏幕上显示:Hello World!
#include < stdio.h>              //包含基本输入输出库文件
int main()                      //主函数名
{
    prinft("Hello World!\n");    //屏幕显示语句
    return 0;                   //表示程序顺利结束
}
```

输出:

```
Hello World!
```

分析:该程序非常简单,仅由一个主函数构成,在主函数中也只有两条语句。

程序的第 1 行是注释。注释以“//”开头,直到该行的末尾,用于说明或解释程序段的功能、变量的作用以及程序员认为应该向程序阅读者说明的其他任何内容。可以看到,在该程序中还有一些注释。在将 C 程序编译成目标代码时所有的注释行都会被忽略,因此即使使用了很多注释也不会影响目标码的效率。恰当地应用注释可以使程序清晰易懂、易于调试,便于程序员之间的交流与协作,所以,在编写程序时,精心撰写注释是一个良好的编程习惯。

第 2 行是编译预处理,把 stdio. h 库文件插入到程序中相应的位置,用来支持函数 printf。

从第 3 行到最后一行是主函数。主函数是该程序的主体部分,由其说明部分

```
int main()
```

和用一对花括号{}括起来的函数体构成。

通常,一个 C 程序包含一个或多个函数,但其中有且只有一个 main 函数,C 程序的执行就是从 main 函数开始的。main 函数左边的关键字 int 表示 main 函数返回一个整数值,这是和程序的倒数第 2 行的 return 0 对应的,有关函数的进一步内容将在第 6 章介绍,这里只要记住每个程序中都要包含这样的语句就行了。

在函数体内,有两条以分号结束的语句:第 1 条是输出语句,用于将字符串显示在计

算机屏幕上；第 2 条是 return 语句，它放在函数的末尾，数值 0 表明程序运行成功。

5.2 使用 Eclipse 和 Visual Studio 编译 C 程序

C 语言的编译器有很多，常见的有 GNU Compiler Collection（或称 GCC）、Microsoft 的 Visual Studio C++ 以及 Turbo C++ 等。本节介绍两种编程环境的使用，其中 Eclipse 仅仅只是一个编辑器，后台需要使用其他的编译器来一起工作。本书选择的是 Eclipse+GCC。

5.2.1 使用 Eclipse 编译 C 程序

在使用 Eclipse 之前，要进行以下的安装设置工作。

(1) J2SE Development Kit(JDK)安装。Eclipse 需要 Java 环境的支持。因此，在安装 Eclipse 之前，需要确保计算机上安装了 JDK，它也是整个 Java 的核心，包括 Java 运行环境(Java Runtime Environment)、Java 工具和 Java 基础的类库。请在 Java 网站(http://www.java.com/zh_CN/)上下载最新版的 Java 并正确安装。

(2) MinGW 安装。MinGW (Minimalist GNU for Windows)是一个自由软件，可以将 C/C++ 撰写的程序编译为 Windows 环境下的可执行文件。它所使用的编译器(如 C 语言的 gcc 或 C++ 语言的 g++ 等)是由 GCC 移植而来。包括了 C、C++、Objective-C、FORTRAN、Java、Ada 等语言的编译器及相关的函式库等必要的文件。GCC 被誉为世界上最重要的软件之一，它由理察·马修·斯托曼(Richard Matthew Stallman，RMS)在 1987 年开始建立，以作为 GNU(GNU's Not UNIX)自由软件计划的编译器。根据 GNU 自由软件的精神，任何人都可以免费取得 GCC 与 MinGW，并且在符合 GNU 通用公共许可证(GNU General Public License，GPL)的情况下，自由地使用、复制、修改和分发 GCC 及 MinGW。MinGW 又称 MinGW32，在不需要第三方(third-party)动态链接函数库(Dynamic Link Library，DLL)支持的情况下，它可以将 C、C++ 等程序编译为可以在 Windows Win32 平台上执行的程序。可以从(http://sourceforge.net/projects/mingw/files/)下载最新版的 MinGW 并安装。对于本书，只需要安装和 C 语言有关的编译器和库即可。

(3) Eclipse 安装。在 Eclipse 的官方网站(https://www.eclipse.org/downloads/)上下载 Eclipse IDE for C/C++ Developers，无须安装，解压缩后直接运行即可。

(4) 设置环境变量。在系统变量中的 Path 添加“MinGW 安装目录\bin;”(例如 C:\MinGW\bin;)。

【例 5-2】 从键盘输入两个整数，计算两个整数的和并输出。

(1) 启动 Eclipse，如图 5-1 所示。

图 5-1 为 C/C++ Eclipse 启动后的界面，如果是第一次启动，需要先选择一个工作文件夹的存放位置。

(2) 选择菜单 File→New→C Project，弹出如图 5-2 所示的对话框，填好项目名称，选择 Empty Project 和 MinGWGCC 后单击 Finish 按钮。

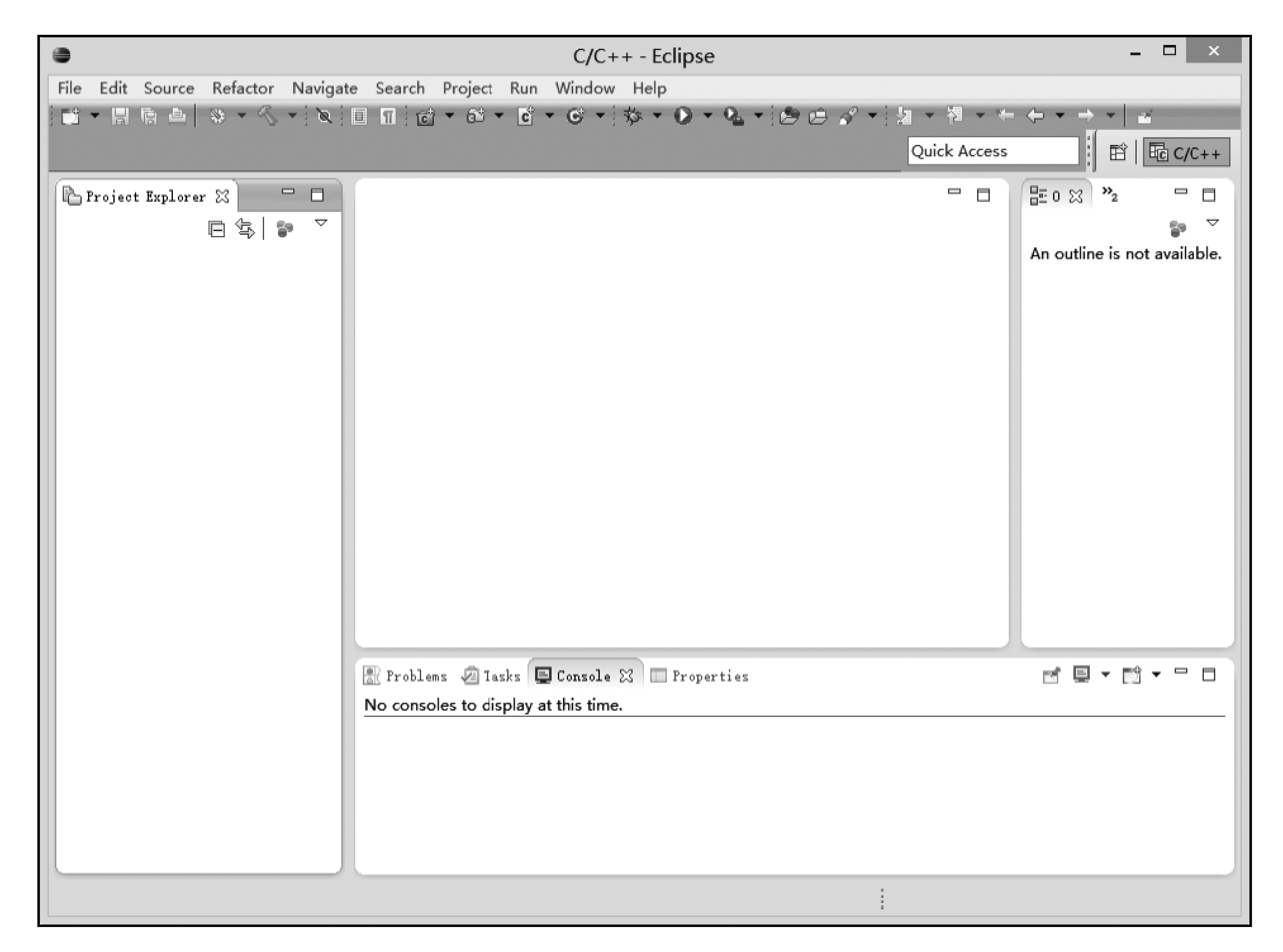


图 5-1　C/C++ Eclipse 启动后的界面

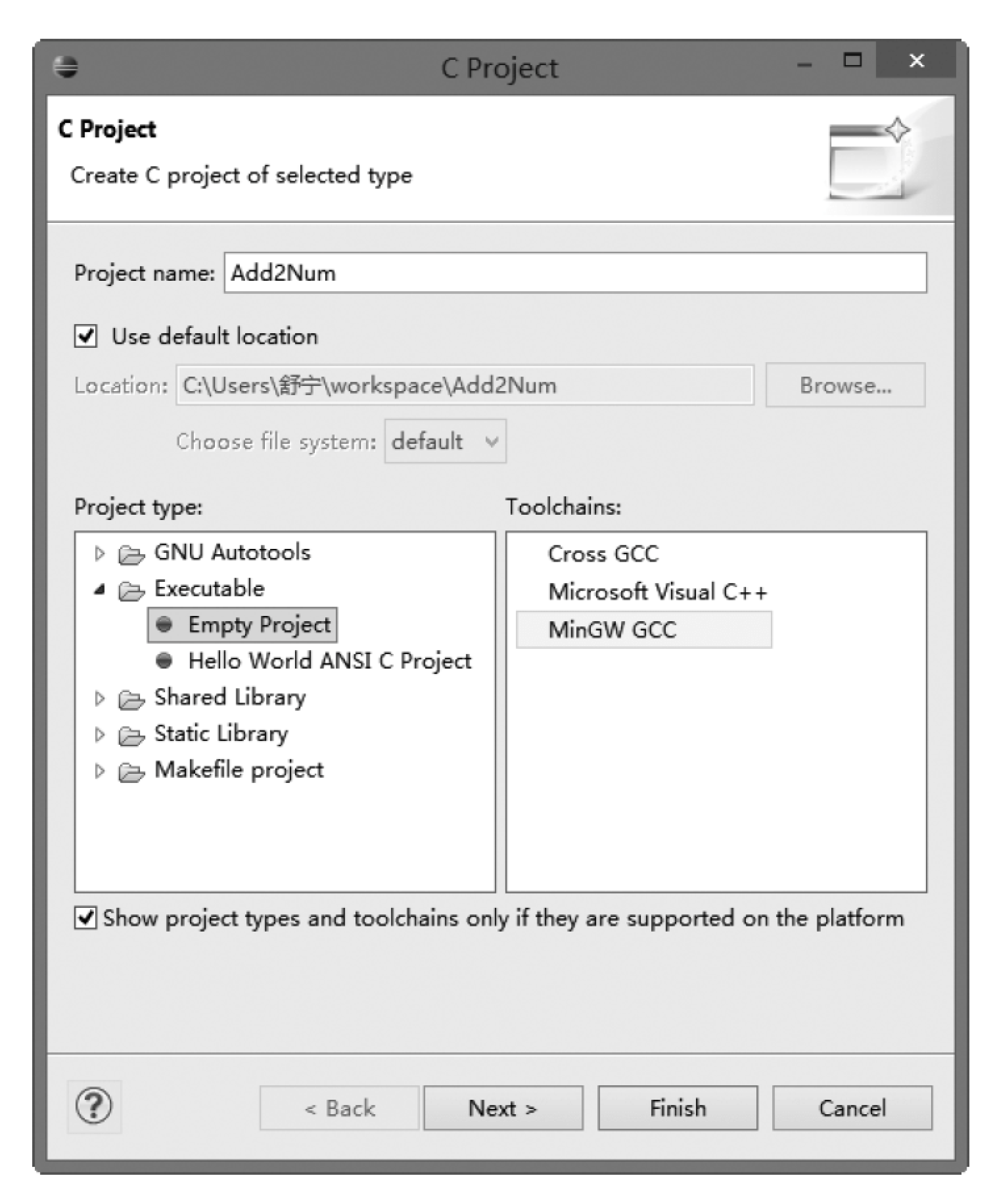


图 5-2　C Project 对话框

图 5-2 Toolchains 中出现的选择项和具体计算机安装了哪些 C/C++ 编译器有关。

(3) 添加一个源代码文件。在 Project Explorer 中的 Add2Num 项目上右击，选择 New→Source File，如图 5-3 所示。在随后弹出的对话框中给源代码文件起一个文件名，注意需要给出文件的后缀名.c。单击 Finish 按钮，如图 5-4 所示。

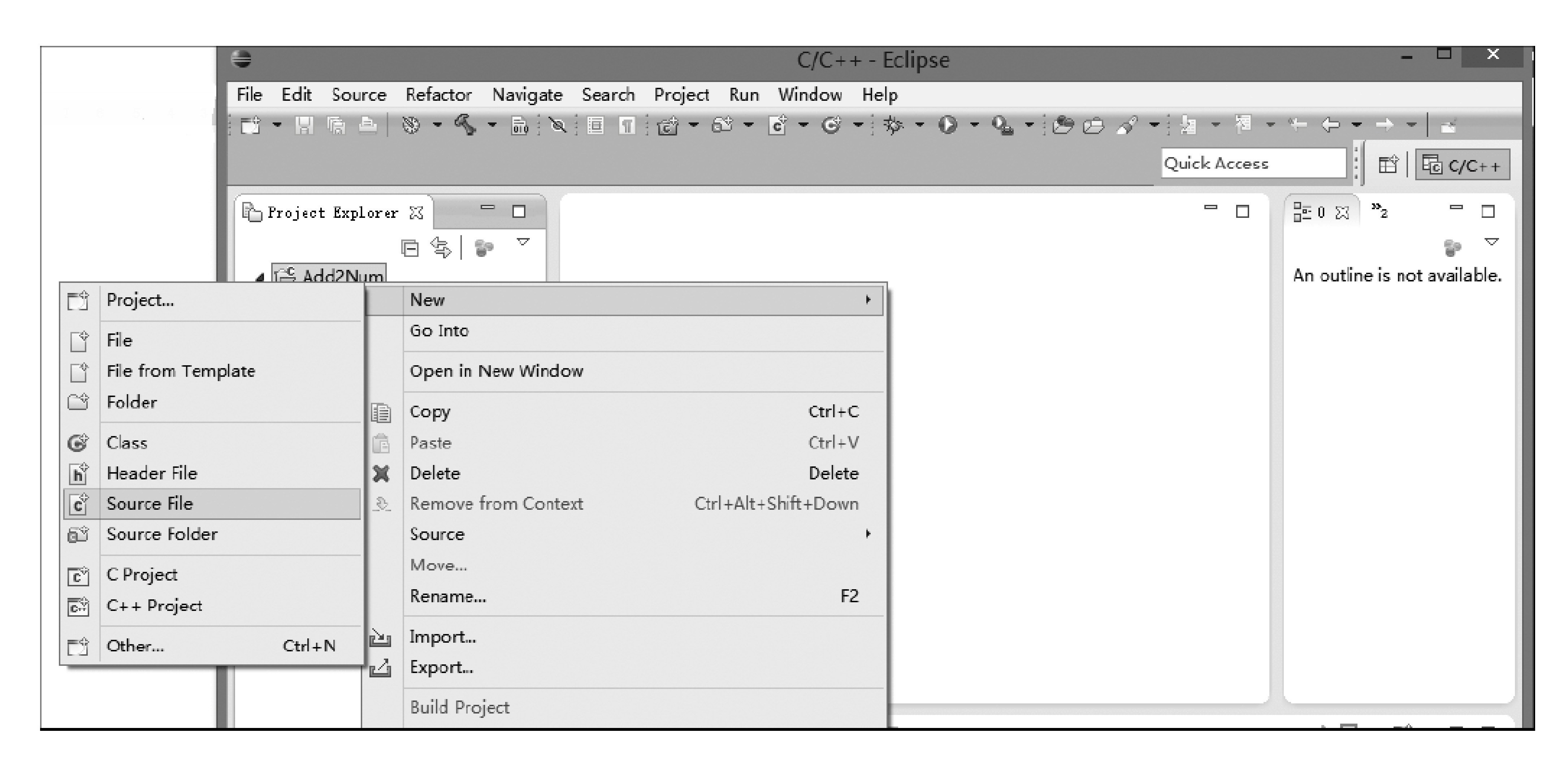


图 5-3 添加一个新的源代码文件

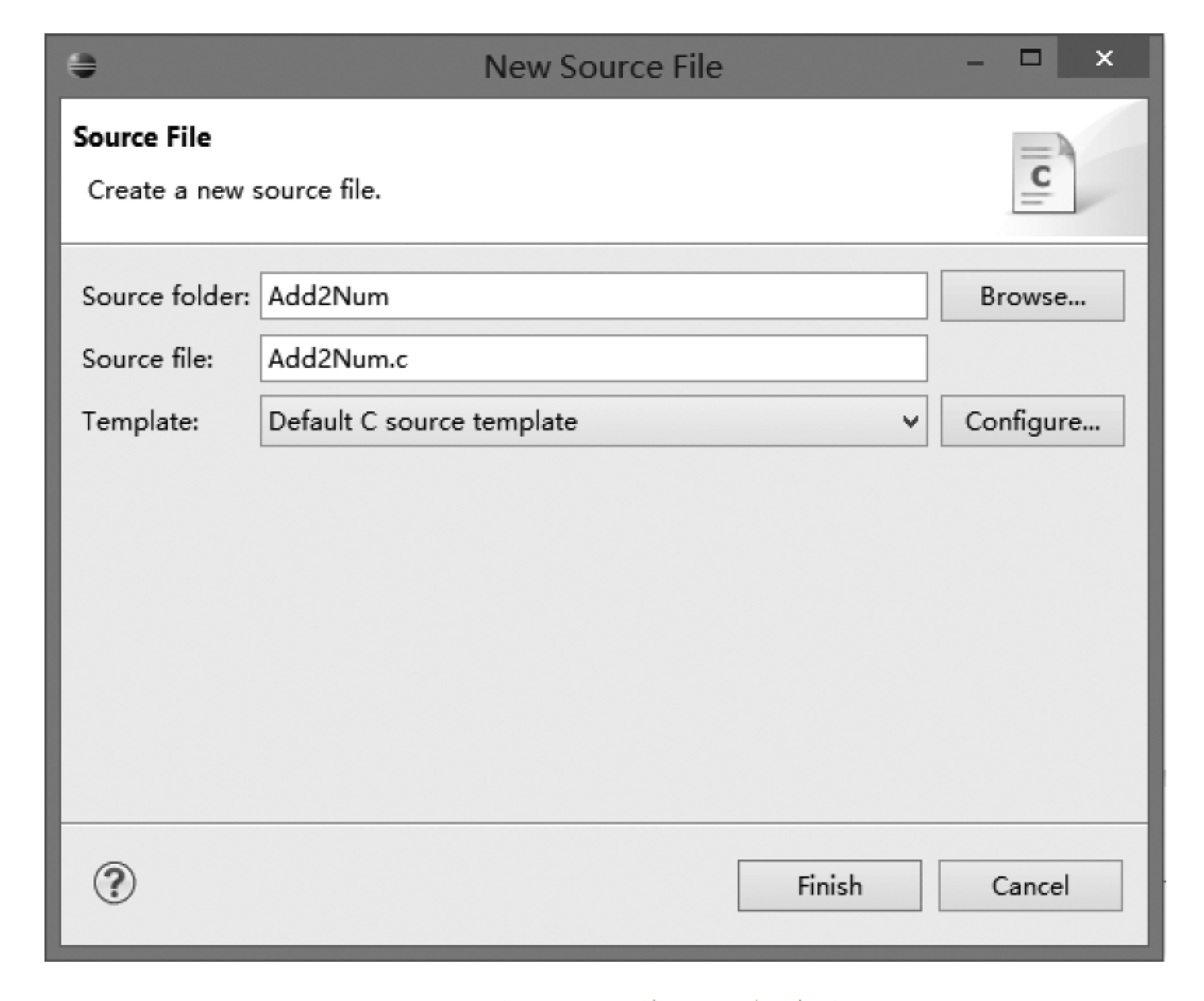


图 5-4 为新的源代码文件命名

(4) 输入代码，如图 5-5 所示。输入完成后单击工具栏上的保存按钮，或者选择菜单 File→Save。

(5) 选择菜单 Project→Build All，将源代码文件编译为可执行文件。然后选择菜单 Run→Run 运行程序，并从 Console 窗口输入两个数，验证程序，如图 5-6 所示。

```
Add2Num.c

 * Add2Num.c
#include <stdio.h>

int main()
{
	int a,b;
	scanf("%d%d",&a,&b);
	printf("The sum is %d",a+b);
	return 0;
}
```

图 5-5　输入程序的源代码

```
Problems  Tasks  Console  Properties
<terminated> Add2Num.exe [C/C++ Application] C:\Users\舒宁\workspace\Add2Num\Debug\Add2Num.exe (14-5-12 下午5:38)
2 3
The sum is 5
```

图 5-6　运行并验证程序的正确性

5.2.2　使用 Visual Studio 编译 C 程序

Visual Studio 是微软公司推出的商业版的编译器，支持多种语言的编译。无论安装 Visual Studio 6.0 还是 Visual Studio 2013，也无论是安装免费的速成版，还是付费的团队版，都可以完成本书的学习。具体的下载安装指导参看微软公司的网站(http://www.visualstudio.com/)。下面以同样的例子使用 Visual Studio 2013 再做一遍。

【例 5-3】　从键盘输入两个整数，计算两个整数的和并输出。

(1) 启动 Visual Studio，选择菜单“文件”→“新建”→“项目”。在弹出的对话框中的左列，依次展开并选择“模板”→Visual C++→Win32，在右侧选择“Win32 控制台应用程序”。在项目名称中填入 Add2Num，然后单击“确定”按钮，如图 5-7 所示。

(2) 在弹出的应用程序向导中，在左侧选择“应用程序设置”，然后在右侧勾选“空项目”复选框，最后单击“完成”按钮，如图 5-8 所示。

(3) 在解决方案资源管理器上展开 Add2Num 项目，在源文件上右击，在快捷菜单中

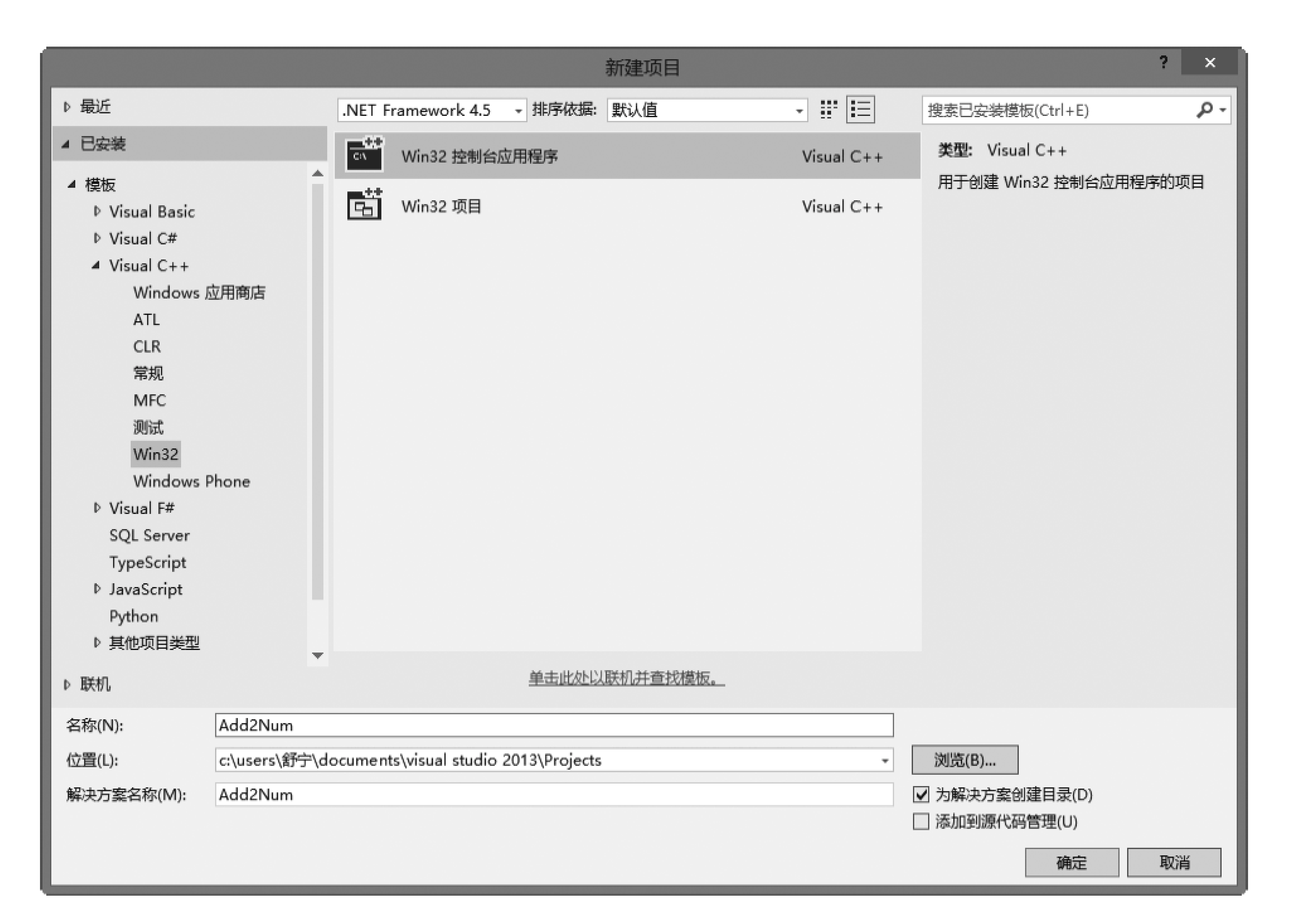


图 5-7　新建一个项目



图 5-8　应用程序向导的设置

选择“添加”→“新建项”,如图 5-9 所示。

(4) 在弹出的对话框中,在“名称”栏填入 Add2Num.c。注意默认的是 C++ 文件,后缀名是.cpp,这里要写入正确的 C 程序的后缀.c,如图 5-10 所示。

(5) 在编辑窗口输入代码,如图 5-11 所示。

(6) 选择菜单“调试”→“开始执行(不调试)”或者按快捷键 Ctrl+F5,运行程序。

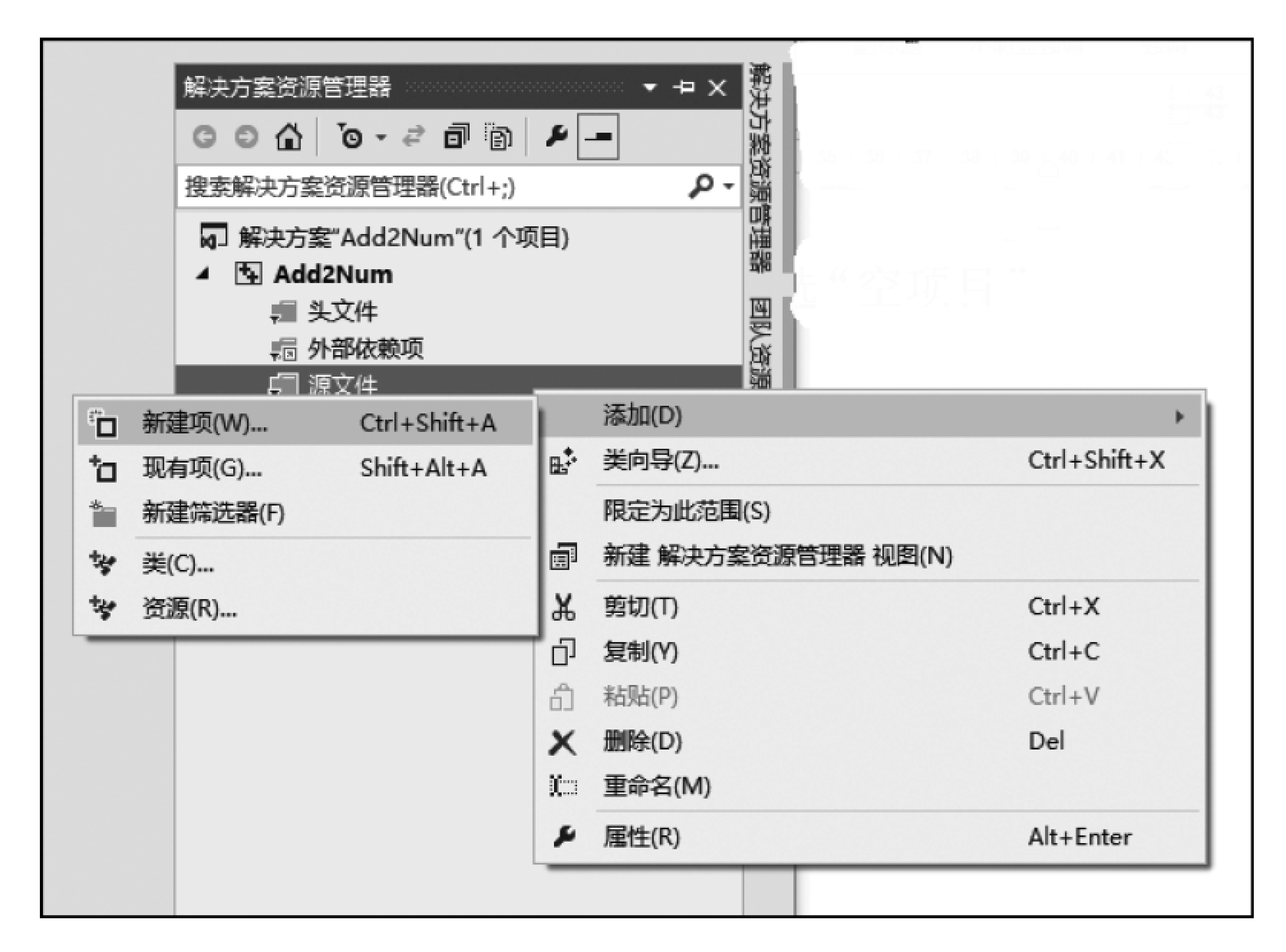


图 5-9 添加一个新的源文件

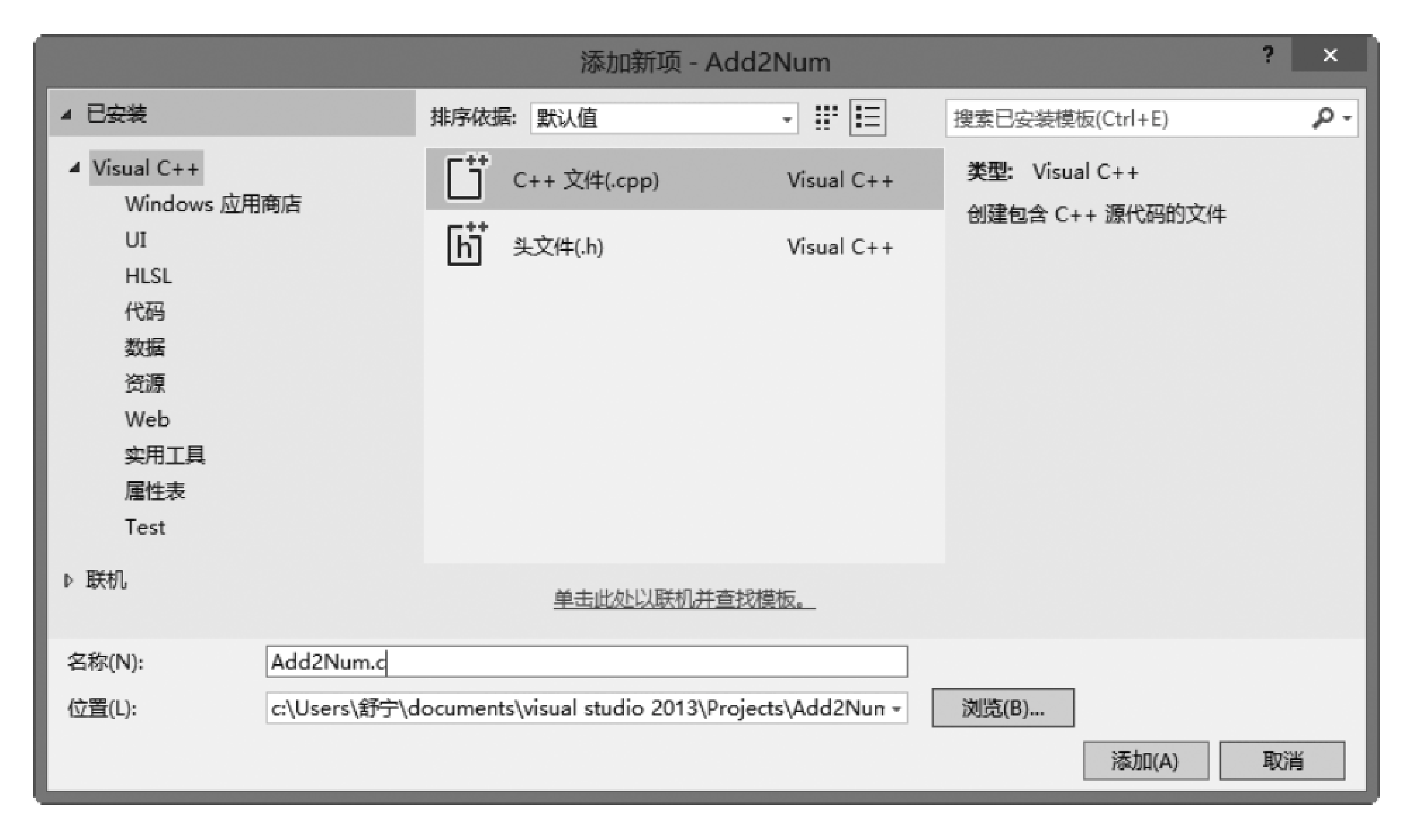


图 5-10 添加一个源代码文件

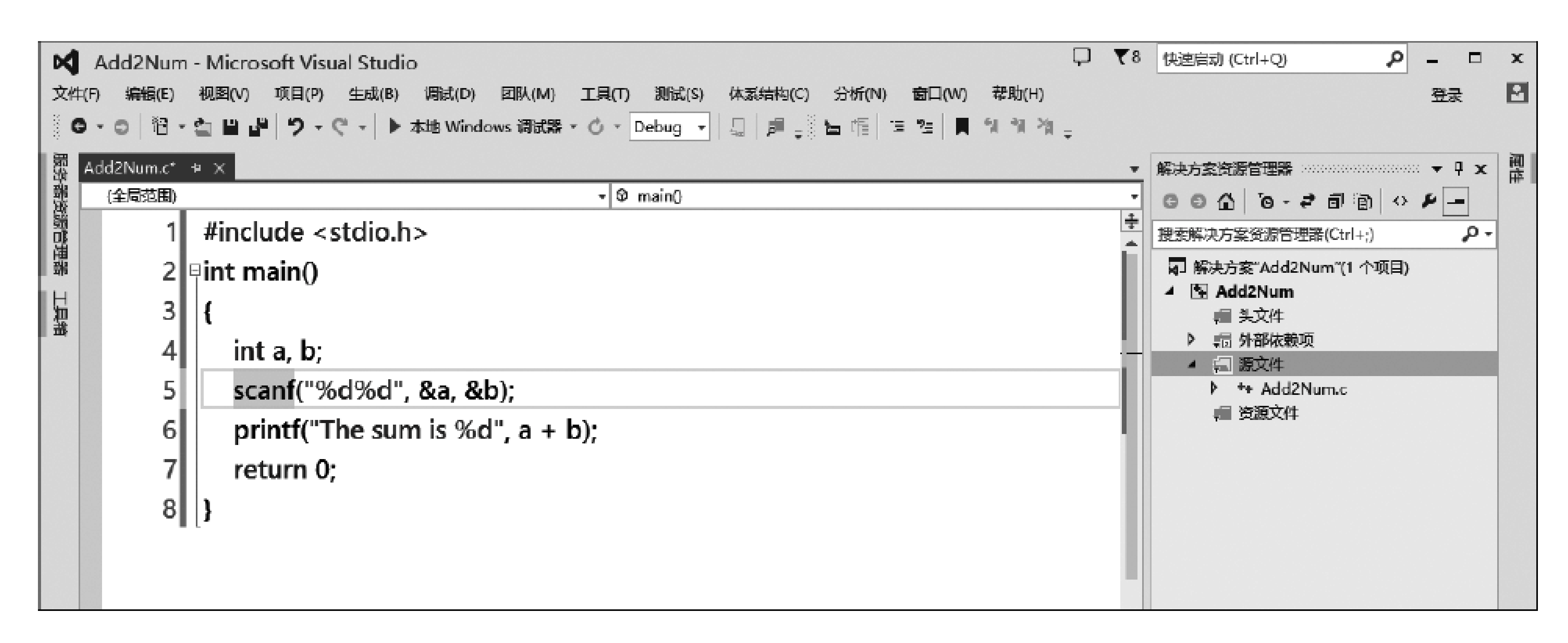


图 5-11 输入代码后的窗口

5.3 输入和输出函数

在前面例子中用到了输入和输出函数 scanf 和 printf，这两个函数分别称为格式输入函数和格式输出函数。其意义是按指定的格式输入和输出值。这两个函数在括号中的参数表都由以下两部分组成：

```
("格式控制串",参数表)
```

格式控制串是一个字符串，必须用双引号括起来，它表示输入输出量的数据类型(参阅 5.5 节)。在 printf 函数中还可以在格式控制串内出现非格式控制字符，这时在显示屏幕上将原文显示。参数表中给出了输入或输出的量。当有多个量时，用逗号间隔，例如：

```
printf("sine of %lf is %lf\n",x,s);
```

其中%lf 为格式字符，表示按双精度浮点数处理。它在格式串中两次出现，对应了 x 和 s 两个变量。其余字符为非格式字符，则照原样输出在屏幕上。

【例 5-4】 各种类型数据的输入和输出。

```
#include <stdio.h>
int main()
{
    int a;                          //整数
    float b;                        //实数
    scanf("%d%f",&a,&b);            //输入
    char c;                         //字符
    scanf("%c",&c);
    char d[20];                     //字符串
    scanf("%s",d);
    //输出
    printf("a=%d,b=%f\n",a,b);
    printf("c=%c,d=%s\n",c,d);
    return 0;
}
```

常用的输入输出格式如下：

- %lf：双精度实数(double)。
- %f：实数(float)。
- %c：字符(char)。
- %s：字符串(char[])。
- %e：科学记数法。
- %g：实数，根据数值的大小，自动选 f 格式、lf 格式或 e 格式(选择输出时占宽度较小的一种)，且不输出无意义的 0。

5.4 C程序的基本要素

5.4.1 C语言字符集、标识符和词汇

字符是组成语言的最基本的元素。C语言字符集由字母、数字、空格、标点和特殊字符组成。在字符常量、字符串常量和注释中还可以使用汉字或其他可表示的图形符号。空格符、制表符、换行符等统称为空白符。空白符只在字符常量和字符串常量中起作用，在其他地方出现时只起间隔作用，编译程序对它们忽略不计。因此在程序中使用空白符与否，对程序的编译不发生影响，但在程序中适当的地方使用空白符将增加程序的清晰性和可读性。

标识符是程序中变量、类型、函数和标号的名称，它可以由程序设计者命名，也可以由系统指定。标识符由字母、数字和下划线"_"组成，第一个字符不能是数字。与FORTRAN和BASIC等程序设计语言不同，C++的编译器把大写和小写字母当作不同的字符，这个特征称为"大小写敏感"。各种C++编译器对在标识符中最多可以使用多少个字符的规定各不相同，ANSI标准规定编译器应识别标识符的前6个字符。在标识符中恰当运用下划线、大小写字母混用以及使用较长的名字都有助于提高程序的可读性。以下标识符是合法的：

```
a,x,x3,BOOK_1,sum5
```

以下标识符是非法的：

3s	（以数字开头）
s * T	（出现非法字符 * ）
－3x	（以减号开头）
bowy-1	（出现非法字符减号）

在使用标识符时还须注意C语言不限制标识符的长度，但它受各种版本的C语言编译系统限制，同时也受到具体机器的限制。例如在某版本C语言中规定标识符前8位有效，当两个标识符前8位相同时，则被认为是同一个标识符。

关键字是由C语言规定的具有特定意义的字符串，通常也称为保留字。用户定义的标识符不应与关键字相同。C语言的关键字分为以下几类：

(1) 类型说明符，用于定义、说明变量、函数或其他数据结构的类型，如int，double等。

(2) 语句定义符，用于表示一个语句的功能。

(3) 预处理命令字，用于表示一个预处理命令，如include。

这些关键字主要有

auto，break，case，char，const，continue，default，do，double，else，enum，extern，float，for，goto，if，int，long，register，return，short，signed，sizeof，static，struct，switch，typedef，

union, unsigned, void, volatile, while, inline, restrict, _ Bool, _ Complex, _ Imaginary, _Alignas, _Alignof, _Atomic, _Static, _assert, _Noreturn, _Thread, _local, _Generic

5.4.2 注释

注释是一种非常重要的机制,没有恰当注释的程序不是一个好程序。

C 语言的注释有两种形式:

(1) //用于单行注释。它以两个反斜杠符起头,直至行末(其实是 C++ 的注释,老的 C 编译器也许不支持这种注释方式)。

(2) /*…*/ 用于多行注释。它可以是用反斜线和星号组合括起的任意文字(注意,这种注释不能互相嵌套)。

注释可以出现在空白符允许出现的任何地方,但习惯上将注释和其所描述的代码相邻,一般可以放在代码的上方或右方,不放在下方。编译器会把注释作为一个空白字符处理。注释应当准确清晰,不要有二义性。恰当使用注释可以使程序容易阅读。

5.4.3 C 源程序结构

一个 C 源程序由一个或多个源文件构成。C 源程序中包括命令、编译指示、说明、定义、语句块和函数等内容。每个源文件可由一个或多个函数组成。一个源程序不论由多少个文件组成,都有一个且只能有一个 main 函数,即主函数。每一个说明、每一个语句都必须以分号结尾。但预处理命令、函数头和花括号"}"之后不能加分号。标识符、关键字之间必须至少加一个空格以示间隔。若已有明显的间隔符,也可不再加空格来间隔。

为了使程序的结构清晰,通常的做法是在一个源文件中放置变量、类型、宏和类等的定义(称为头文件,无后缀或后缀为.h),然后在另一个源文件说明引用这些变量(称为源程序文件,后缀一般为.c)。采用这种方式编写的程序,很容易查找和修改各类定义。

文件包含是编译预处理命令中的一种,它是指一个程序将另一个指定文件的内容包含进来,即将另一个程序文件在编译时嵌入到本文件中。源程序中可以有预处理命令(include 命令仅为其中的一种),通常应放在源文件或源程序的最前面。

文件包含操作的一般格式为

```
#include <文件名>
```

或者

```
#include "文件名"
```

其中"文件名"是指被嵌入的 C 源程序文件的文件名,必须用双引号或者尖括号(实际上是一个小于号和一个大于号)括起来。通过使用不同的括号可以通知预处理程序在查找嵌入文件时采用不同的策略。如果使用了尖括号,那么预处理程序在系统规定的目录(通常是在系统的 include 子目录)中查找该文件。如果使用双引号,那么编译预处理程序首先在当前目录中查找嵌入文件,如果找不到则再去由操作系统的 path 命令所设置的

各个目录中去查找。如果仍然没有查找到，最后再去上述规定的目录(include 子目录)中查找。

原来的源程序文件和用文件包含命令嵌入的源程序文件在逻辑上被看成是同一个文件，经过编译后生成一个目标代码文件，如图 5-12 所示。

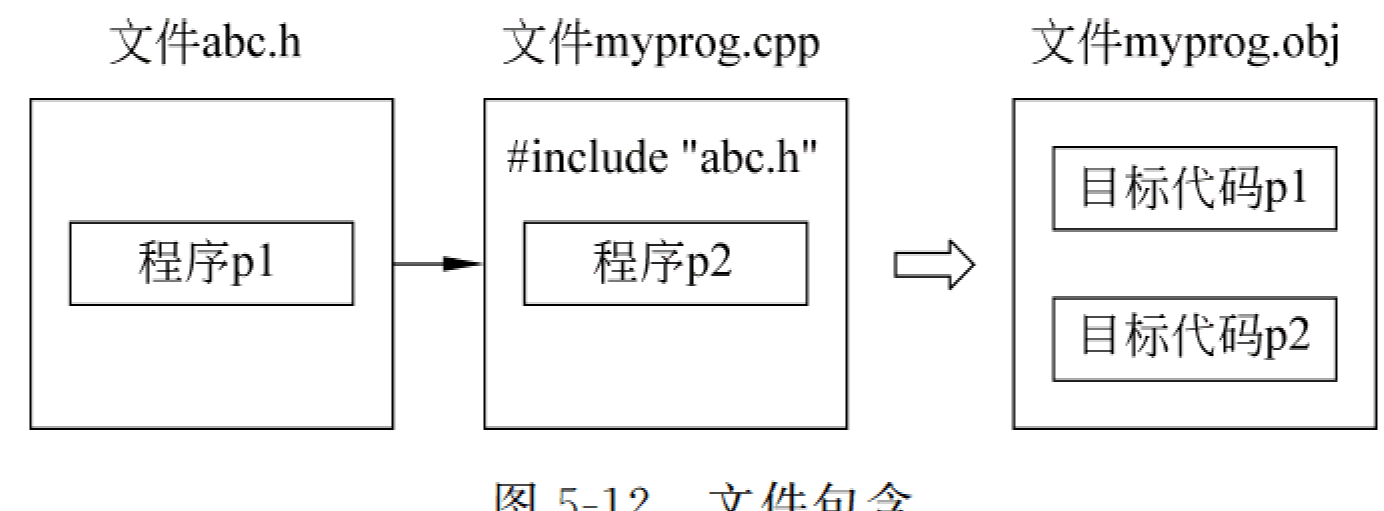


图 5-12　文件包含

从书写清晰，便于阅读、理解和维护的角度出发，在书写程序时应遵循以下规则：

(1) 一个说明或一个语句占一行。

(2) 用{}括起来的部分，通常表示了程序的某一层次结构。{}一般与该结构语句的第一个字母对齐，并单独占一行。

(3) 低一层次的语句或说明可比高一层次的语句或说明缩进若干格后书写，以便看起来层次更加清晰，增加程序的可读性。

在编程时应力求遵循这些规则，以养成良好的编程风格。

5.5 数据类型

程序的主要任务是对数据进行处理，而数据有多种类型，如数值数据、文字数据、图像数据以及声音数据等，其中最基本的也是最常用的是数值数据和文字数据。

无论什么数据，计算机在对其进行处理时都要先存放在内存中。显然，不同类型的数据在存储器中存放的格式也不相同，甚至同一类数据，有时为了具体问题的处理方便起见，也可以使用不同的存储格式。例如，数值数据的存储格式又可以分为整型、长整型、浮点型和双精度型等几种类型，文字数据也可以分为单个字符和字符串。因此在程序中对各种数据进行处理之前都要对其类型(也就是存储格式)预先加以说明，这样做一是便于为这些数据分配相应的存储空间，二是说明了程序处理数据时应采用何种具体运算方法。

C 语言的数据有两种基本形式：常量和变量。常量的用法比较简单，通过本身的书写格式就说明了该常量的类型；而在程序中使用变量之前必须先说明其类型，否则程序无法为该变量分配存储空间。也就是说，变量要“先说明，后使用”。这条原则不仅适合变量，同样适合 C 程序的其他成分，如函数、类型和宏等。

C 语言的一个主要特点是它的数据类型相当丰富，不但有字符型、短整型、整型、长整型、浮点型和双精度型等基本数据类型以及由它们构成的数组，还可以通过构造类型描述较复杂的数据对象。C 语言的数据类型如图 5-13 所示。在本节中主要介绍几种基本数据类型的说明和使用方法。

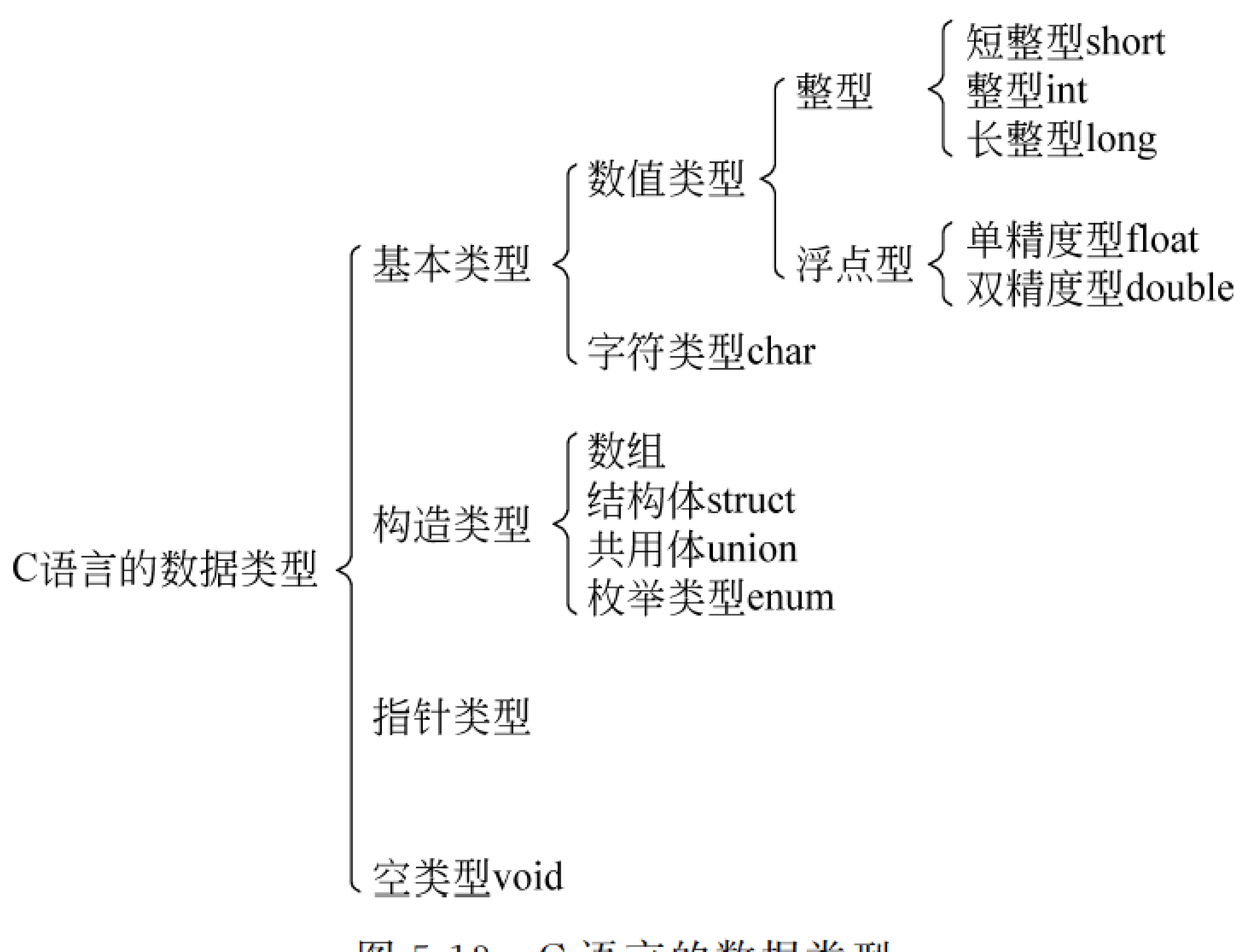
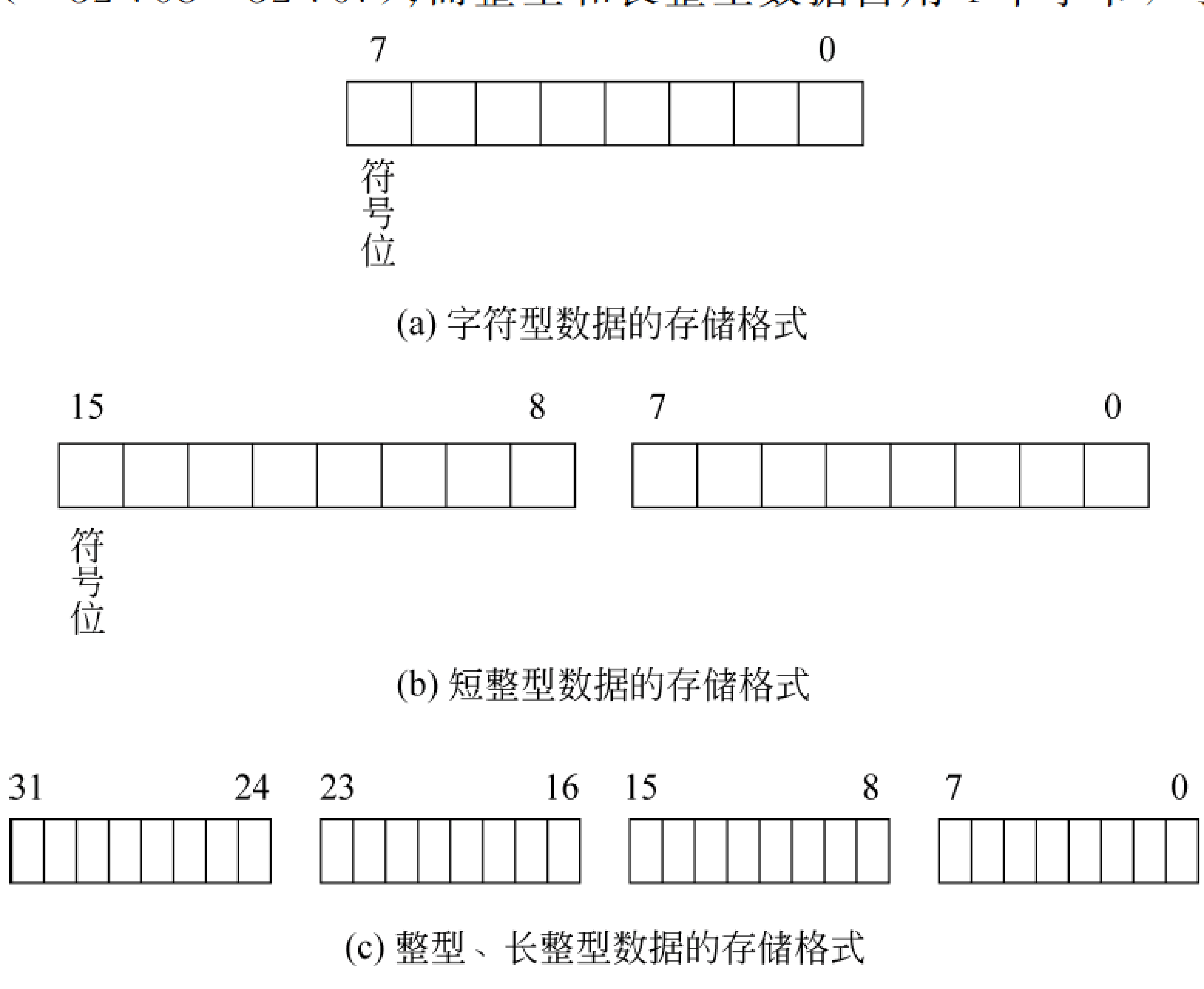


图 5-13　C 语言的数据类型

在 C 语言中，存放一个整数数据可以使用字符型、短整型、整型和长整型 4 种类型。这 4 种类型的格式相似，其最高位均为符号位，0 表示正值，1 表示负值。

字符型数据占用一个字节存储空间，短整型数据占用两个字节，整型和长整型数据要占用 4 个字节的存储空间①，见图 5-14。字符型数据占用一个字节，共 8 个二进制位，其中第 7 位是符号位，因此数值部分可用 7 个二进制位表示，即字符型可以表现的数值范围为 $-2^7 \sim 2^7-1$（$-128 \sim 127$）；同理，短整型数据占用 2 个字节，可以表示的数值范围为 $-2^{15} \sim 2^{15}-1$（$-32\,768 \sim 32\,767$）；而整型和长整型数据占用 4 个字节，可以表示的数值



(a) 字符型数据的存储格式

(b) 短整型数据的存储格式

(c) 整型、长整型数据的存储格式

图 5-14　整型数据的 3 种存储格式

① 在不同的系统中，每个数据类型所占的存储字节数目可能有所不同，因此在使用某个版本的 C 编译器之前，应该仔细阅读其用户手册，或使用 sizeof 运算符弄清其数据长度等基本参数。

范围为$-2^{31}\sim2^{31}-1$。

在编写程序时应根据数据的实际情况选用相应的数据类型。一般的整数数据,大多选用整型表示。至于字符型,因其表示范围太小,通常很少用其存放整型数据,而是用来存放字符的代码。

在日常生活或工程实践中,大多数数据既可以取整数数值,也可以取带有小数部分的非整数数值,例如人的身高和体重、货物的金额等。

浮点类型使用了4个字节存放数据,所以其精度有限,一般只有7位有效数字,可以表示的数值范围为$-3.4\times10^{-38}\sim3.4\times10^{38}$。有时可能需要进行精度更高的计算,这时可以使用双精度类型。双精度类型数据共占用8个字节,其有效数字可达15位,取值范围约为$-1.7\times10^{-308}\sim1.7\times10^{308}$。浮点类型的存储格式如图5-15所示。

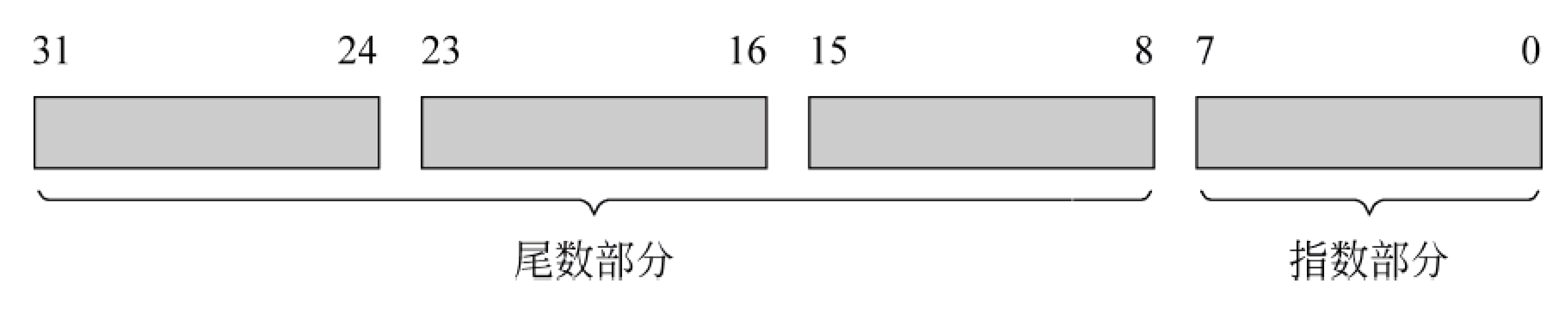


图5-15　浮点型数据的存储格式

5.5.1　常量

常量是指在程序运行的整个过程中其值始终不可改变的量。C语言中常用的有4种常量:整型常量、实型常量、字符常量和字符串常量。常量在表达方式上既可以直接表示,如常量1,3.14,'A',"Hello"等,分别表示整数1、实数3.14、字符A和字符串Hello。也可用符号代表,如用PI代表圆周率3.14159,直接表示的常量称为直接常量,用符号代表的常量称为符号常量。

符号常量在使用之前必须先定义,其一般形式为

#define　标识符 常量

其中#define也是一条预处理命令(预处理命令都以#开头),称为宏定义命令(在后面将进一步介绍),其功能是把该标识符定义为其后的常量值。一经定义,以后在程序中所有出现该标识符的地方均代之以该常量值。习惯上符号常量的标识符用大写字母,变量标识符用小写字母,以示区别。

【例5-5】　符号常量的使用。

```
#include <stdio.h>
#define PRICE 60
int main()
{
    int num,total;
    num=10;
```

```
    total=num* PRICE;
    printf("total=%d",total);
    return 0;
}
```

程序中用标识符 PRICE 代表一个常量,称为符号常量。符号常量与变量不同,它的值在其作用域内不能改变,也不能再被赋值。使用符号常量可以做到代码含义清楚,也能做到“一改全改”。

1. 整型常量

整型常量就是整常数。在 C 语言中,使用的整常数有八进制、十六进制和十进制 3 种。十进制整常数没有前缀。以下各数是合法的十进制整常数:

237,−568,65535,1627

以下各数不是合法的十进制整常数:

023(不能有前导 0),23D(含有非十进制数码)

在程序中是根据前缀来区分各种进制数的。因此在书写常数时不要把前缀弄错造成结果不正确。

八进制整常数必须以 0 开头,即以 0 作为八进制数的前缀。数码取值为 0~7。八进制数通常是无符号数。例如:

015(十进制为 13),0101(十进制为 65),0177777(十进制为 65535)

十六进制整常数的前缀为 0x 或 0X。其数码取值为 0~9、A~F 或 a~f。例如:

0x2A(十进制为 42),0xA0(十进制为 160),0xFFFF(十进制为 65535)

如果要指明一个整数数值使用长整型格式存放,可以在数值之后写一个字母 l 或 L。由于小写 l 很容易和数字 1 相混,建议使用大写字母 L 表示长整形常数。例如,以下为十进制长整常数:

158L(十进制为 158),358000L(十进制为 358000)

八进制长整常数:

012L(十进制为 10),077L(十进制为 63),0200000L(十进制为 65536)

十六进制长整常数:

0x15L(十进制为 21),0xA5L(十进制为 165),0x10000L(十进制为 65536)

无符号数也可用后缀表示,整型常数的无符号数的后缀为 u 或 U。例如:

358u,0x38Au,235Lu

均为无符号数。

2. 实型常量

在 C 语言中,可以使用浮点类型表示这类数据。浮点数据类型使用科学记数法表示数值:将数值分为尾数部分和指数部分,前者是一个纯小数,且小数点后第 1 位不为 0;后者是一个整数值。这两部分均可以为正或为负。实际数值等于尾数部分乘上 10 的指数部分的幂次。例如,圆周率 π 可以写成

0.3141593×10¹

C 语言的浮点类型常数可以使用两种方式书写，一种是小数形式，例如：

0.0，1.0，－2.68，3.141593，－637.312，32767.0，－32768.0

这时应注意，即使浮点类型的常数没有小数部分也应补上“.0”，否则会与整型常数混淆。另一种是科学记数形式，其中用字母 e 或者 E 表示 10 的幂次，例如：

0.0E0，6.226e－4，6.226E4，－1.267E－20

C 语言允许浮点数使用后缀。后缀为 f 或 F 即表示该数为浮点数。如 356f 和 356. 是等价的。

3. 字符常量

在 C 语言中，文字数据有两种：单个的字符和字符串。对于字符数据来说，实际上存储的是其编码。由于英语中的基本符号较少，只有 52 个大小写字母、10 个数字、空格和若干标点符号，再加上一些控制字符，如回车、换行、蜂鸣器等，总共约 100 个，因此，可以使用一个整数表示某个字符的代码。目前最常用的代码标准是 ASCII 码，ASCII 码共使用了 128 个编码，分别使用整数 0～127 表示，可以参看附录 B。

一般来说，在用 C 语言编写程序时，单个的字符变量多选用整型变量存放，因为其数目有限，占用存储空间不多，而现在计算机的 CPU 中的数据字长多为 16 位以上，使用整型的运算速度比较快。但是对于字符串数据，由于占用的存储空间比较多，所以均选用字符型数组存放，一个数组元素（字符类型的变量）正好存放一个字符的 ASCII 码。

字符型常量实际上就是单个字符的 ASCII 码。但是在程序中直接使用码值很不直观，例如从码值 48 和 97 很难看出它们实际上代表的是字符'0'和'a'。因此在 C++ 语言中引入了一套助记符号来表示 ASCII 码。对于字母、数字和标点符号等可见字符来说，其助记码就是在该符号两边加上单引号。例如：

'a'，'A'，'1'，' '，'＋'

另外，还有一些字符是比较特殊的(可能不可显示或无法通过键盘输入)，如控制字符、引号和反斜杠符等，对此 C++ 专门提供了一种称为转义序列的表示方法。它使用由一个反斜杠符和一个符号组成的转义字符表示这些特殊字符，例如：

'\n'(换行)，'\r'(回车)，'\t'(横向跳格)，'\''(单引号)

常用的转义字符见表 5-2。

表 5-2　常用的转义字符

转义字符	含　　义
\n	换行符
\r	回车符
\t	制表符
\f	换页符
\b	退格符
\\	反斜杠
\'	单引号

转义字符	含　　义
\"	双引号
\0	0
\nnn	码值为 nnn 的 ASCII 码,nnn 表示 3 位八进制数
\xhh	码值为 hh 的 ASCII 码,hh 表示 2 位十六进制数

注意：上述助记符实际上仍是一个整数，因此也可以参加运算。例如：

```
c='A'+2;                        // c被赋值为字母 C
if(x>='0' && x<='9')            // 如果 x是一数字的 ASCII 码
    x=x-'0';                    // 将其转换为相应的数值
```

另外还需要注意的是：字符常量只能用单引号括起来，不能用双引号或其他括号；字符常量只能是单个字符，不能是字符串；字符可以是字符集中的任意字符；特别要注意'5'和 5 是不同的。

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示。表 5-1 中的\nnn 和\xhh 正是为此而提出的。nnn 和 hh 分别为八进制和十六进制的 ASCII 代码。如\101 表示字母 A，\102 表示字母 B，\134 表示反斜线，\x0A 表示换行等。

【例 5-6】 转义字符的使用。

```
#include<stdio.h>
int main()
{
    printf("  ab  c\tde\rf\n");
    printf("hijk\tL\x40M\n");
    return 0;
}
```

运行结果：

```
  ab  c  de
f
hijk L@M
```

4. 字符串常量

字符串常量是用双引号括起来的一串字符，例如：

"Visual C++ "，"12.34"，"This is a string.\n"

字符串常量在内存占用的实际存储字节数要比字符串中的字符个数多 1 个，即在字符串的尾部还要添加一个数值为 0 的字符，用于表示字符串的结束。该字符也可以使用转义序列'\0'表示。以字符串"MONDAY"为例，其实际存储形式见图 5-16。

'M'	'O'	'N'	'D'	'A'	'Y'	'\0'

图 5-16　字符串的存储方式

因此,'B'与"B"是有区别的,前者是一个字符常量,而后者是字符串常量,由两个字符'B'和'\0'组成。

5.5.2 变量

与常量相反,变量是指在程序运行期间可以改变的量。每个变量都要有一个名字,即变量名,用标识符来表示。变量在内存中占据一定的存储单元,并在该存储单元中存放变量的值。变量分为不同的类型,如整型变量、双精度变量、字符变量等。

如果要在程序中使用变量,就必须先对变量进行声明,即要使用变量说明语句。C 语言的数据变量说明语句的格式为

<类型说明符><变量名 1>[,<变量名 2>,…,<变量名 n>];

其中,类型说明符指出了变量的数据类型,该类型决定了变量的格式和行为。

下面是一些基本数据类型的类型说明符:

```
short  <短整型变量表>
int  <整型变量表>;
long  <长整型变量表>;
char  <字符类型变量表>;
float  <浮点类型变量名表>;
double  <双精度类型变量名表>;
```

下面列出几个变量说明语句的例子:

```
char c1,c2;                    //说明了两个字符型变量
int i,j,k;                     //说明了 3个整型变量
long len;                      //说明了一个长整型变量
float average,sum;             //说明了两个浮点类型的变量
double distance,weight;        //说明了两个双精度类型的变量
```

C 语言允许在说明变量的同时对变量赋一个初值,例如:

```
int count=0;
double pi=3.14159265358979E0;
int upper='A';
```

【例 5-7】 字符'A'的不同赋值方法。用不同的方法给字符变量赋值,结果都输出字符'A'。

分析:可以直接用字符'A'进行赋值,也可以用其对应的 ASCII 码赋值,还可以使用表达式。

程序:

```
#include<stdio.h>
int main()
{
```

```
    char c;
    int n=20;
    c='A';
    printf("直接使用字符赋值的结果 %c\n",c);
    c=65;
    printf("使用字符的 ASCII 码的结果 %c\n",c);
    c='\x41';
    printf("使用十六进制 ASCII 码的结果 %c\n",c);
    c='\101';
    printf("使用八进制 ASCII 码的结果 %c\n",c);
    c=n+45;
    printf("使用表达式的值作为 ASCII 码的结果 %c\n",c);
}
```

5.5.3 类型修饰符

C 语言提供的修饰符如下：

```
signed(有符号)
unsigned(无符号)
```

signed 的意义为带符号。由于基本类型 char、short、int、long 等均为带符号位的类型，所以 signed 修饰符的用途不大。unsigned 适用于 char、short、int 和 long 4 种整数类型，其意义为取消符号位，只表示正值。这样，unsigned char 的表示范围就变为 0～255，unsigned short 类型的表示范围变为 0～65 535，而 unsigned int （可以直接写成 unsigned）类型和 unsigned long 类型的表示范围变为 $0\sim2^{32}-1$。

当类型修饰符应用于 int 类型之前时，可以省略 int 不写(即 int 是隐含表示的)。例如：

```
signed int      等价于   signed
unsigned int    等价于   unsigned
```

实际上，前面所讲的 long 和 short 也是类型修饰符，只不过是省略了后面的 int 罢了。如果将 long 用于 double 之前，会形成一种新的数据类型：long double，而且在有些系统中它可以提供比 double 类型更多的存储空间。

5.6 运算符和表达式

表达式是由运算符将运算对象(如常数、变量和函数等)连接起来的具有合法语义的式子。在 C 语言中，由于运算符比较丰富(达数十种之多)，因而可以构成灵活多样的表达式。这些表达式的应用一方面可以使程序编写得短小简捷，另一方面还可以完成某些在其他高级程序设计语言中较难实现的运算功能。

学习C语言的表达式时应注意以下几个方面：

（1）运算符的正确书写方法。C语言的许多运算符与通常在数学公式中所见到的符号有很大差别，例如整除求余（%）、相等（==）、逻辑运算与（&&）等。

（2）运算符的确切含义和功能。C语言中有些很特殊的运算符，要理解其准确的含义。

（3）运算符与运算对象的关系。C语言的运算符可以分为单目运算符（仅对一个运算对象进行操作）、双目运算符（需要两个运算对象），甚至还有复合表达式，其中的两个运算符对3个或者更多个运算对象进行操作。

（4）运算符具有优先级和结合方向。如果一个运算对象的两边有不同的运算符，首先执行优先级别较高的运算。如果一个运算对象两边的运算符级别相同，则应按由左向右的方向顺序处理。各运算符的优先顺序可以参看表5-4。如果编程序时对运算符的优先顺序没有把握，可以通过使用括号来明确其运算顺序。

5.6.1 算术运算符和算术表达式

C语言的算术运算符有

+（加），-（减），*（乘），/（除），%（整除求余）

其中/为除法运算符。注意：如果除数和被除数均为整型数据，则结果也是整数。例如，5/3的结果为1。%为整除求余运算符。%运算符两侧均应为整型数据，其运算结果为两个运算对象做除法运算的余数。例如，5%3的结果为2。

在C语言中，不允许两个算术运算符紧挨在一起，也不能像在数学运算式中那样任意省略乘号以及用中圆点“·”代替乘号等。如果遇到这些情况，应该使用括号将连续的算术运算符隔开，或者在适当的位置上加上乘法运算符。例如：

x*-y　　应写成　x*(-y)

(x+y)(x-y)　应写成　(x+y)*(x-y)

5.6.2 关系运算符和关系表达式

关系运算符又称比较运算符，C语言中有6种关系运算符：

>（大于），<（小于），==（等于），>=（大于等于），<=（小于等于），!=（不等于）

用关系运算符将两个表达式连接起来就构成了关系表达式。关系表达式的值为0（表达式为假）或者为1（表达式为真），例如：

```
x>=3
a+b==c
```

注意，算术运算符的优先级高于关系运算符，即

a+b==c　等价于　(a+b)==c

5.6.3 逻辑运算符和逻辑表达式

简单的关系比较是不能满足实际的编程需要的，一般还需要用逻辑运算符将关系表达式或逻辑量连接起来，构成较复杂的逻辑表达式。逻辑表达式的值也是逻辑量。

C 语言中提供了 3 种逻辑运算符：

!(逻辑非)，&&(逻辑与)，‖(逻辑或)

在逻辑运算符中，逻辑与 && 的优先级高于逻辑或 ‖ 的优先级，而所有的比较运算符的优先级均高于以上两个逻辑运算符。至于逻辑非运算符!，由于这是一个单目运算符，所以和其他单目运算符（例如用于作正、负号的＋和－）一样，优先级高于包括算术运算符在内的所有双目运算符。例如，表达式

```
x* y>z && x* y<100 || -x* y>0 && !isgreat(z)
```

的运算顺序为[①]：

```
计算 x* y                              //算术运算优先于比较运算
计算 x* y>z                            //比较运算优先于逻辑运算
计算 x* y<100                          //比较运算优先于逻辑运算
计算 x* y>z && x* y<100                //逻辑与运算优先于逻辑或运算
计算 -x                                //单目运算优先于双目运算
计算 -x* y                             //算术运算优先于比较运算
计算 -x* y>0                           //比较运算优先于逻辑运算
计算 isgreat(z)                        //计算函数值优先于任何运算符
计算 !isgreat(z)                       //单目运算优先于双目运算
计算 -x* y>0 && !isgreat(z)            //逻辑与运算优先于逻辑或运算
计算 x* y>z && x* y<100||-x* y>0 && !isgreat(z)
```

5.6.4 赋值运算符和赋值表达式

C 语言将赋值作为一个运算符处理。赋值运算符为＝，用于构造赋值表达式。赋值表达式的格式为

V=e

其中 *V* 表示变量，*e* 表示一个表达式。赋值表达式的值等于赋值运算符右边的表达式的值。和其他表达式一样，赋值表达式也可以作为更复杂的表达式的组成部分。例如：

```
i=j=m* n;
```

---

① 实际的运算顺序与这里介绍的会略有差别，这是因为 C 语言在执行表达式时进行了优化。例如，对于表达式 x * y>z && x * y<100 来说，如果第一个比较表达式 x * y>z 不成立，则无论第二个表达式 x * y<100 成立或不成立，整个表达的值均为 0(不成立)，因此就无须计算第二个表达式。

由于赋值运算符的优先级较低(仅比逗号运算符高)。并列的赋值运算符之间的结合方向为从右向左,所以上述语句的执行顺序是:首先计算出表达式 m * n 的值,然后再处理表达式 j=m * n,该表达式的值就是 m * n 的值,将该值存入变量 j。最后,处理表达式 i=j=m * n,其值即第一个赋值运算符右面的整个表达式的值,因此也就是 m * n 的值。将第一个赋值运算符右面整个表达式的值存入变量 i。因此,上述表达式语句的作用是将 m * n 的值赋给变量 i 和 j。整个运算过程如下(设 m 的值为 2,n 的值为 3):

计算 m * n 的值:2 * 3 等于 6。

计算 j=m * n 的值:j=6 的值等于 6,将 6 存入变量 j。

计算 i=j=m * n 的值:i=6 的值等于 6,将 6 存入变量 i。

5.6.5 自增运算符和自减运算符

C 语言中有两个很有特色的运算符:自增运算符++和自减运算符--。这两个运算符也是 C 程序中最常用的运算符,以至于它们几乎成为 C 程序的象征。

++和--运算符都是单目运算符,其运算对象常为整型变量或指针变量。这两个运算符既可以放在作为运算对象的变量之前,也可以放在变量之后,但对运算对象的值影响不同。++和--这两个运算符真正的价值在于它们和赋值运算符类似,在参加运算的同时还改变了作为运算对象的变量的值。++i 和 i++会使变量 i 的值增大 1;类似地,--i 和 i--会使变量 i 的值减 1。++和--构成的 4 种表达式的含义见表 5-3(设 i 为一个整型变量)。

表 5-3 自增运算符和自减运算符的用法

表达式	表达式的值	副作用	表达式	表达式的值	副作用
i++	i	i 的值增大 1	i--	i	i 的值减小 1
++i	i+1	i 的值增大 1	--i	i-1	i 的值减小 1

++表达式和--表达式既可以单独使用,也可以出现于更复杂的表达式中。例如:

```
i++;                    //i 增加 1
--i;                    //i 减少 1
x=array[++i];           //将 array[i+1]的值赋给 x,并使 i 增加 1
s1[i++]=s2[j++];        //将 s2[j]赋给 s1[i],然后分别使 i 和 j 增加 1
```

作为运算符来说,++和--的优先级较高,高于所有算术运算符和逻辑运算符。但在使用这两个运算符时要注意它们的运算对象只能是变量,不能是其他表达式。例如,(i+j)++就是一个错误的表达式。

引入含有++、--以及赋值运算符这类表达式的目的在于简化程序的编写。例如,表达式语句 i=j=m * n;的作用和以下两条语句完全一样:

```
j=m* n;
i=j;
```

而表达式语句 s1[i++]=s2[j++]；其实正是下列语句的简化表达方式：

```
s1[i]=s2[j];
i=i+1;
j=j+1;
```

5.6.6 问号表达式和逗号表达式

C 语言中还提供了一种比较复杂的表达式，即问号表达式，又称条件表达式。问号表达式使用两个运算符(?和:)对 3 个运算对象进行操作，格式为

<表达式 1>?<表达式 2>:<表达式 3>

问号表达式的值是这样确定的：如果<表达式 1>的值为非零值，则问号表达式的值就是<表达式 2>的值；如果<表达式 1>的值等于 0，则问号表达式的值为<表达式 3>的值。利用问号表达式可以简化某些选择结构的编程。例如，以下的分支语句

```
if(x>y)
    z=x;
else
    z=y;
```

等价于语句

```
z=x>y?x:y;
```

【例 5-8】 编写一个求绝对值的函数。

```
double mydabs(double x)
{
    return x>0?x:-x;
}
```

在 C 语言中可以使用逗号(,)将几个表达式连接起来，构成逗号表达式。逗号表达式的格式为

<表达式 1>,<表达式 2>,…,<表达式 n>

在程序执行时，按从左到右的顺序执行组成逗号表达式的各表达式，而将最后一个表达式（即表达式 n）的值作为逗号表达式的值。

逗号表达式常用于简化程序的编写。例如，如下程序结构

```
if(x>y)
{
    t=x;
    x=y;
    y=t;
```

```
    }
```

可以利用逗号表达式简化为

```
    if(x>y)
        t=x,x=y,y=t;
```

【例 5-9】 已知两个变量 x=1,z=100,输入第 3 个数给变量 y,判断 y 的值是否在 x 和 z 之间。

```
#include<stdio.h>
int main()
{
    int x,y,z;
    x=1;
    z=100;
    printf("请输入变量 y的值\n");
    scanf("%d",&y);
    printf((y>x)+(y<z)==2?"y的值在 x和 z之间":"y的值不在 x和 z之间");
    return 0;
}
```

*5.6.7 位运算表达式

与大多数程序设计语言不同,C 语言还提供了位运算功能。所谓位运算,就是直接对数据中的最小单位——二进制位进行操作。C 语言中位运算的操作对象只能是各种整型(如 char 型、int 型、unsigned 型以及 long 型等)数据。位运算符共有以下几种。

1. 按位与(&)

两个整型数据中的二进制位做“与”运算。“与”运算的规则为:如果参加运算的两个二进制位均为 1,则结果为 1,否则结果为 0。例如:

```
int x=3,y=5;
```

则 x 值对应的二进制表示为 00000000 00000011,y 值对应的二进制表示为 00000000 00000101。按位与 x&y 的运算过程为

```
    00000000 00000011
  & 00000000 00000101
  -------------------
    00000000 00000001
```

因此 x&y 的结果为二进制数 1,换算成十进制也是 1。按位与运算常用于屏蔽数据中的某些位。

【例 5-10】 取一个整型变量的最低 4 位。

取整型变量的最低 4 位,只需将其与二进制数 0000 0000 0000 1111 作按位与运算。而二进制数 0000 0000 0000 1111 转换为十六进制数是 0x000F。因此,要取某整型量的

最低 4 位,可以将其与十六进制数 0x000F 作按位与运算。

程序:

```
//宏 tran16():取整型量的最低 4 位
#define tran16(x) ((x)&0x0f)
```

分析:由于取整型变量的低 4 位的算法非常简单,所以将其编写为带参数的宏比较合适(宏会在后面详细讲述)。由于带参宏对参数的类型不敏感,所以这个宏可以用于各种整型量。

2. 按位或(|)

两个整型数据中的二进制位做"或"运算。"或"运算的规则为:只要参加运算的两个二进制位中有一个为 1,则结果就是 1;只有在参加运算的两个二进制位均为 0 的情况下结果才是 0。例如:

```
int x=3,y=5;
```

则 x 值对应的二进制表示为 00000000 00000011,y 值对应的二进制表示为 00000000 00000101。按位或 x|y 的运算过程为

```
  00000000 00000011
| 00000000 00000101
-------------------
  00000000 00000111
```

因此 x|y 的结果为二进制数 111,换算成十进制则是 7。按位或运算常用于将多个数据内容拼接在一起。

3. 按位异或(^)

两个整型数据中的二进制位做"异或"运算。"异或"运算的规则为:如果参加运算的两个二进制位不同,则运算结果为 1;相同,则结果为 0。例如:

```
int x=3,y=5;
```

则 x 值对应的二进制表示为 00000000 00000011,y 值对应的二进制表示为 00000000 00000101。按位异或 x^y 的运算过程为

```
  00000000 00000011
^ 00000000 00000101
-------------------
  00000000 00000110
```

因此 x^y 的结果为二进制数 110,换算成十进制则是 6。按位异或运算有一个有趣的性质,即在同一数据上两次异或一个值,结果变回原来的值。例如,在 3 和 5 异或的结果 6 上再次异或 5,则会得到原来的数值 3:

```
  00000000 00000110
^ 00000000 00000101
-------------------
  00000000 00000011
```

异或运算的这个性质在编制动画程序时特别有用。

4. 按位取反(～)

按位取反是单目运算符,只需一个运算对象。按位取反运算将作为运算对象的整型数据中的二进制位做“求反”运算。“求反”运算的规则很简单:如果原来的二进制位为1,则运算结果为0,否则结果为1,即运算结果和原来的数据相反。例如:

```
int x=3;
```

则x值对应的二进制表示为00000000 00000011。按位求反～x的运算过程为

```
～ 00000000 00000011
─────────────────────
   11111111 11111100
```

因此～x的结果为二进制数11111111 11111100,换算成十六进制则是0xfffc。在设计图像处理程序时经常要用到按位求反运算。

按位取反运算符(～)的优先级为2,比大多数算术运算、关系运算和逻辑运算中的双目运算符以及其他位运算符的优先级别要高。

5. 左移位运算符(<<)

左移位运算用于将整型数据中的各个二进制位全部左移若干位,并在该数据的右端添加相同个数的0。例如:

```
int x=3;
```

则x值对应的二进制表示为00000000 00000011。将x左移3位可以通过以下语句实现:

```
x=x<<3;
```

其运算过程为

```
     00000000 00000011
<<                  11
──────────────────────
     00000000 00011000
```

因此x<<3的结果为二进制数00000000 00011000,换算成十六进制则是0x0018。左移位运算常和按位或运算一起使用,用于将两个数据的内容拼在一起。

6. 右移位运算符(>>)

右移位运算用于将整型数据中的各个二进制位全部右移若干位,并在该数据的左端添加相同个数的0。例如:

```
int x=255;
```

则x值对应的二进制表示为00000000 11111111。将x右移4位可以通过以下语句实现:

```
x=x>>4;
```

其运算过程为

```
    00000000 11111111
>>                100
    00000000 00001111
```

因此 x>>4 的结果为二进制数 00000000 00001111,换算成十进制则是 15。右移位运算常和按位与运算一起使用,用于从一个数据中分离出某些位来。

7. 位运算复合赋值运算符

与算术赋值运算符类似,由位运算符与赋值运算符也可以组成位运算复合赋值运算符:

&=, |=, ^=, >>=, <<=

例如:

```
a &=b  等价于  a=a & b
a<<=2  等价于  a=a<<2
```

【例 5-11】 使用异或运算交换两个整型变量的值。

分析:位运算中异或运算^的规则时,对应的二进制位相同为 0,不同为 1。可以得出结论,对于两个变量 x 和 y,x=x^y^y。利用这个关系可以在交换两个变量的值时不借助第 3 个变量。

程序:

```
#include <stdio.h>
int main()
{
    int n1,n2;
    printf("请输入两个整数\n");
    scanf("%d%d",&n1,&n2);
    printf("交换前\n");
    printf("n1=%d\tn2=%d\n",n1,n2);
    n1^=n2^=n1^=n2;
    printf("交换后\n");
    printf("n1=%d\tn2=%d\n",n1,n2);
    return 0;
}
```

5.6.8 表达式中各运算符的运算顺序

四则运算的运算顺序可以归纳为“先乘除,后加减”,也就是说乘除运算的优先级别比加减运算的优先级别要高。C 语言中有几十种运算符,仅用一句“先乘除,后加减”是无法表示各种运算符之间的优先关系的,因此必须有更严格的确定各运算符优先关系的规则。表 5-4 列出了各种运算符的优先级别和同级别运算符的运算顺序(结合方向)。

表 5-4 运算符的优先级别和结合方向

<table>
<tr><th>优先级别</th><th>运 算 符</th><th>运算形式</th><th>结合方向</th><th>名称或含义</th></tr>
<tr><td>1</td><td>()
[]
.
-></td><td>(e)
a[e]
x.y
p->x</td><td>自左至右</td><td>圆括号
数组下标
结构体成员
用指针访问结构体成员</td></tr>
<tr><td>2</td><td>- +
++ --
!
~
(t)
*
&
sizeof</td><td>-e
++x 或 x++
!e
~e
(t)e
*p
&x
sizeof(t)</td><td>自右至左</td><td>负号和正号
自增运算和自减运算
逻辑非
按位取反
类型转换
由地址求内容
求变量的地址
求某类型变量的长度</td></tr>
<tr><td>3</td><td>* / %</td><td>e1*e2</td><td>自左至右</td><td>乘、除和求余</td></tr>
<tr><td>4</td><td>+ -</td><td>e1+e2</td><td>自左至右</td><td>加和减</td></tr>
<tr><td>5</td><td><< >></td><td>e1<<d2</td><td>自左至右</td><td>左移和右移</td></tr>
<tr><td>6</td><td>< <= > >=</td><td>e1<e2</td><td>自左至右</td><td>关系运算(比较)</td></tr>
<tr><td>7</td><td>== !=</td><td>e1==e2</td><td>自左至右</td><td>等于和不等于比较</td></tr>
<tr><td>8</td><td>&</td><td>e1&e2</td><td>自左至右</td><td>按位与</td></tr>
<tr><td>9</td><td>^</td><td>e1^e2</td><td>自左至右</td><td>按位异或</td></tr>
<tr><td>10</td><td>|</td><td>e1|e2</td><td>自左至右</td><td>按位或</td></tr>
<tr><td>11</td><td>&&</td><td>e1&&e2</td><td>自左至右</td><td>逻辑与(并且)</td></tr>
<tr><td>12</td><td>||</td><td>e1||e2</td><td>自左至右</td><td>逻辑或(或者)</td></tr>
<tr><td>13</td><td>? :</td><td>e1?e2:e3</td><td>自右至左</td><td>条件运算</td></tr>
<tr><td rowspan="2">14</td><td>=</td><td rowspan="2"></td><td rowspan="2">自右至左</td><td>赋值运算</td></tr>
<tr><td>+= -= *=
/= %= >>=
<<= &= ^=
|=</td><td>复合赋值运算</td></tr>
<tr><td>15</td><td>,</td><td>e1,e2</td><td>自左至右</td><td>顺序求值运算</td></tr>
</table>

说明:"运算形式"一栏中各字母的含义为:a,数组;e,表达式;p,指针;t,类型;x、y,变量。

由表 5-4 可以看出,运算优先级的数字越大,优先级别越低。优先级别最高的是括号,所以如果要改变混合运算中的运算次序,或者对运算次序把握不准时,可以使用括号来明确规定运算的顺序。

运算符的结合方向是对级别相同的运算符而言的,说明了在几个并列的级别相同的运算符中运算的次序。大部分运算符的结合方向都是"自左至右",例如表达式 x*y/3,运算次序就是先计算 x*y,然后将其结果除以 3。也有些运算符的结合顺序与此相反,是"自右至左",例如赋值运算符,在表达式 i=j=0 中,计算顺序就是首先将 0 赋给变量 j,然

后再将表达式 j=0 的值(仍为 0)赋给变量 i。

5.6.9 不同类型数据之间的混合算术运算

大多数运算符对运算对象的类型有严格的要求。例如,%运算符只能用于两个整型数据的运算,所有的位运算符也只适用于整型数据。但是算术四则运算符适用于所有的整型(包括 char、int 和 long)、浮点型(float)和双精度型(double)数据,因此存在一个问题:不同类型的数据的运算结果的类型怎样确定?

C 语言规定,不同类型的数据在参加运算之前会自动转换成相同的类型,然后再进行运算。运算结果的类型也就是转换后的类型。转换的规则如下。

(1) 级别低的类型转换为级别高的类型。各类型按级别由低到高的顺序为 char、int、unsigned、long、unsigned long、float、double。

例如,一个 char 类型的数据和一个 int 类型的数据运算,结果为 int 型;一个 int 型的数据和一个 double 型数据的运算,结果的类型为 double 型。

另外,C 语言规定,有符号类型数据和无符号类型的数据进行混合运算,结果为无符号类型。例如,int 型数据和 unsigned 型数据的运算结果为 unsigned 型。

对于赋值运算来说,如果赋值运算符右边的表达式的类型与赋值运算符左边的变量的类型不一致,则赋值时会首先将赋值运算符右边的表达式按赋值运算符左边的变量的类型进行转换,然后将转换后的表达式的值赋给赋值运算左边的变量。整个赋值表达式的值及其类型也是这个经过转换后的值及其类型。例如:

```
float x;
int i;
x=i=3.1416;
```

则变量 i 的值为 3,并且赋值表达式 i = 3.1416 的类型为 int,值也是 3。因此尽管变量 x 的类型为 float,但对其赋值的结果是,x 的值为 3.0 而不是 3.1416。上述赋值表达式语句实际上完全相当于以下两个赋值表达式语句的效果:

```
i=3.1416;
x=i;
```

(2) 可以使用强制类型转换。在程序中使用强制类型转换操作符可以明确地控制类型转换。强制类型转换操作符由一个放在括号中的类型名组成,置于表达式之前,其结果是表达式的类型被转换为由强制类型转换操作符所标明的类型。例如,如果 i 的类型为 int,表达式 (double)i 将 i 强制转换为 double 类型。

算术表达式的强制类型转换的最主要的用途是防止丢失整数除法结果中的小数部分。例如:

```
int i1=100,i2=40;
double d1;
d1=i1/i2;
```

这段程序的结果是 double 类型的变量 d1 的内容被赋值为 2.0，虽然 100/40 求值应为 2.5。其原因是表达式 i1/i2 包含了两个 int 类型的变量，该表达式的类型当然也应该是 int 类型。因此，它只能表示整数部分，结果中的小数部分就丢失了。虽然将 i1/i2 的结果赋值给了一个双精度类型的变量，但这时结果中的小数部分已经被丢掉了。

为了防止这种误差，其中一个 int 类型的变量必须强制转换为 double 类型：

```
d1=(double)i1/i2;
```

在这种情况下，变量 i1 先被强制转换为 double 类型，另一个变量 i2 就被自动地转换为 double 类型，并且整个表达式的类型也是 double，结果的小数部分就会被保留。

【例 5-12】 输入通话的开始时间和结束时间，然后计算通话的秒数。

分析：通话时间分别输入时、分和秒，为简化问题，假定开始时间和结束时间都在同一天内。计算时先分别计算开始时间和结束时间相对该天零点零分零秒的总秒数，然后再相减。

程序：

```
#include<stdio.h>
int main()
{
    int h1,m1,s1,t1;                    //开始时间
    int h2,m2,s2,t2;                    //结束时间
    int t;                              //通话时间
    printf("请输入开始通话的时间,时分秒之间使用空格、回车键或 Tab 键\n");
    scanf("%d%d%d",&h1,&m1,&s1);
    printf("请输入结束通话的时间,时分秒之间使用空格、回车键或 Tab 键\n");
    scanf("%d%d%d",&h2,&m2,&s2);
    t1=h1*3600+m1*60+s1;
    t2=h2*3600+m2*60+s2;
    t=t2-t1;
    printf("通话时间为:%d秒\n",t);
    return 0;
}
```

5.6.10 typedef 语句

typedef 语句（类型说明语句）的功能是为某个已有的数据类型定义一个新的同义字或别名。其格式为

typedef <数据类型或数据类型名><新数据类型名>

值得注意的是，typedef 语句不能创建任何新的数据类型，它只能为一种现存的数据类型创建一个别名。

例如，为 float 类型取一个别名 real 可以使用类型说明语句：

```
typedef float real;
```

此后的程序中可以使用 real 代替 float 说明浮点型变量：

```
real x,y;
```

等价于语句

```
float x,y;
```

5.6.11 运算符与表达式例题

【例 5-13】 根据三边长求三角形面积。

算法：利用海伦公式：$A=\sqrt{s(s-a)(s-b)(s-c)}$，其中 a、b、c 分别为三角形 3 条边的长度，$s=\frac{1}{2}(a+b+c)$。

程序：

```
#include <stdio.h>
#include <math.h>
int main()
{
    double a,b,c,s,area;
    printf("Please input a,b,c = ");
    scanf("%lf%lf%lf",&a,&b,&c);
    s= (a+ b+ c)/2;
    area= sqrt(s * (s- a) * (s- b) * (s- c));
    printf("area= %lf\n",area);
    return 0;
}
```

输入：

```
3  4  5
```

输出：

```
area= 6
```

分析：为简单起见，程序未考虑对数据的检验，即未检查输入的 3 边长是否能构成一个三角形。实际上，数据检验是程序的重要组成部分，应予以足够的重视。

【例 5-14】 输入一个四位无符号整数，反序输出这四位数的 4 个数字字符。

算法：从输入的无符号整数 n 中依次分解出个位数字、十位数字、百位数字、千位数字并依次存放到变量 c1、c2、c3、c4 中，如将 n%10 的值即个位数字存入 c1 中，将 n/10%10 的值即十位数字存入 c2 中，将 n/100%10 的值即百位数字存入 c3 中，将 n/1000 的值即千位数字存入 c4 中。再将各数字值+'0'则转为对应的数字字符。

程序：

```
#include <stdio.h>
int main()
{
    unsigned int n;
    char c1,c2,c3,c4;
    printf("Please input one integer between 1000 and 9999: \n");
    scanf("%d",&n);
    printf("Before inverse the number is: %d\n",n);
    c1=n%10+'0';              //分离个位数字
    c2=n/10%10+'0';           //分离十位数字
    c3=n/100%10+'0';          //分离百位数字
    c4=n/1000+'0';            //分离千位数字
    printf("After inverse the number is: %c%c%c%c",c1,c2,c3,c4);
    return 0;
}
```

输入和输出：

```
Please input one integer between 1000 and 9999:
1234
Before inverse the number is: 1234
After inverse the number is: 4321
```

【例 5-15】 求一元二次方程 $ax^2+bx+c=0$ 的根，其中系数 a、b、c 为实数，由键盘输入。

算法：设 $\Delta=b^2-4ac$，知道当 $\Delta=0$ 时，方程有一个重根；当 $\Delta>0$，方程有两个不同的实根；如果 $\Delta<0$，则有两个共轭的复根。

程序：

```
#include <stdio.h>
#include <math.h>
int main()
{
    double a,b,c,delta,p,q;
    printf("Please intput a,b,c=");
    scanf("%lf%lf%lf",&a,&b,&c);
    delta=b*b-4*a*c;
    p=-b/(2*a);
    q=sqrt(fabs(delta))/(2*a);
    if(delta>=0)
    {
        printf("x1=%lf\nx2=\n",p+q,p-q);
    }
    else
```

```
    {
        printf("x1=%lf+j%lf\n",p,q);
        printf("x2=%lf-j%lf\n",p,q);
    }

}
```

【例 5-16】 温度转换：输入一个华氏温度，计算并输出对应的摄氏温度值。

算法：温度的转化公式是 $C=5(F-32)/9$。

程序：

```
#include<stdio.h>
int main()
{
    double c,f;
    printf("请输入一个华氏温度：");
    scanf("%lf",&f);
    c=5.0/9.0*(f-32);
    printf("对应于华氏温度%lf的摄氏温度为%lf\n",f,c);
    return 0;

}
```

【例 5-17】 大小写转换：输入一个字符，判断它是否为大写字母，如果是，将其转换为对应的小写字母输出；否则，不用转换直接输出。

算法：ASCII 表中的大写字母 A～Z 是连续排列的，小写字母 a～z 也是连续排列的，但大写字母和小写字母并没有排在一起。因此，如果一个字符是大写字母，就可以通过对其 ASCII 码作如下运算转换为对应的小写字母的 ASCII 码：

小写字母＝大写字母－'A'＋'a'

程序：

```
#include<stdio.h>
int main()
{
    char ch;
    printf("请输入一个字母：");
    scanf("%c",&ch);
    if(ch>='A' && ch<='Z')
      ch=ch-'A'+'a';
    printf("将大写转换为小写后,该字母为：%c\n",ch);
    return 0;
}
```

【例 5-18】 找零钱问题：假定有伍角、壹角、伍分、贰分和壹分共 5 种硬币，在给顾客找硬币时，一般都会尽可能选用硬币个数最少的方法。例如，当要给某顾客找七角二分钱

时，会给他一个伍角、2个壹角和1个贰分的硬币。请编写一个程序，输入的是要找给顾客的零钱(以分为单位)，输出的是应该找回的各种硬币数目，并保证找回的硬币数最少。

算法：每次尽可能选择面值最大的硬币即可。

程序：

```
/#include <stdio.h>
int main()
{
    int change;            //存放零钱的变量
    printf("请输入要找给顾客的零钱(以分为单位):");
    scanf("%d",&change);
    printf("找给顾客的伍角硬币个数为:%d\n",change/50);
    change=change%50;
    printf("找给顾客的壹角硬币个数为:%d\n",change/10);
    change=change%10;
    printf("找给顾客的伍分硬币个数为:%d\n",change/5);
    change=change%5;
    printf("找给顾客的贰分硬币个数为:%d\n",change/2);
    change=change%2;
    printf("找给顾客的壹分硬币个数为:%d\n",change);
    return 0;
}
```

【例5-19】 判断一个四位整数是否为回文数。

分析：回文数是指由该数各位上数字反序构成的数与原数相同，对于四位整数，可以简单地判断两个条件即千位和个位、百位和十位是否相等，所以先分解出各位数字。

程序：

```
#include <stdio.h>
#include <math.h>
int main()
{
    int n,d1,d2,d3,d4;            //d1到d4分别用来表示各位数字
    printf("请输入一个四位整数:)";
    scanf("%d",&n);
    d1=n/1000;                    //千位
    d2=n/100%10;                  //百位
    d3=n/10%10;                   //十位
    d4=n%10;                      //个位
    if(d1==d4 && d2==d3)
        printf("该数是回文数\n");
    else
        printf("该数不是回文数\n");
}
```

5.7 控制结构

5.7.1 顺序结构

在用 C 语言编写程序时，实现顺序结构的方法非常简单：只需将两个语句顺序排列即可。如交换两个整数的值的程序段

```
r=p;
p=q;
q=r;
```

就是顺序结构。

5.7.2 选择结构

C 语言的选择结构是通过 if-else 语句实现的。其格式为：

```
if(<表达式>)
    <程序模块 1>;
else
    <程序模块 2>;
```

一般来说，“程序模块 1”和“程序模块 2”可以是各种语句，甚至包括 if-else 语句和后面要介绍的循环语句。如果“程序模块 1”和“程序模块 2”比较复杂，不能简单地用一条语句实现时，需要使用由一对花括号“{}”括起来的程序段落。如果仅有 1 条语句，则花括号可以省略(建议初学者即使只有 1 条语句，也不要省略花括号)：

```
if(<表达式>)
{
    ……
}
else
{
    ……
}
```

这种用花括号括起来的程序段落又称为分程序。分程序是 C 语言的一个重要概念。具体说来，一个分程序具有下述形式：

```
{
    <局部数据说明部分>
    <执行语句段>
}
```

即分程序是由花括号括起来的一组语句。当然，分程序中也可以再嵌套新的分程序。分程序是C程序的基本单位之一。

分程序在语法上是一个整体，相当于一个语句。因此分程序可以直接和各种控制语句结合使用，用以构成C程序的各种复杂的控制结构。在分程序中定义的变量的作用范围仅限于该分程序内部。

在if语句中用＜表达式＞的值来判断程序的流向，如果＜表达式＞的值不为0，表示条件成立，此时执行＜语句1＞；否则(即＜表达式＞的值等于0)执行＜语句2＞。作条件用的表达式中通常含有比较运算符或逻辑运算符，例如：

```
x>y                   //x大于 y则表达式的值非 0,否则表达式的值为 0
x>=0.0 && x<=1.0      //x的值在 0和 1之间则表达式的值非 0,否则为 0
```

其中的逻辑运算符 && 表示“并且”。这类表达式在其中的比较或逻辑运算的结果为真时取值1，为假时取值0，因此正好可以用来在if语句中表示条件。

只有一个分支的选择结构可以使用不含else部分的if语句表示：

```
if(<表达式>)
    <语句>;
```

或者

```
if(<表达式>)
{
    ……
}
```

即，如果＜表达式＞的值不为0时执行＜语句＞或分程序，否则直接执行if语句后面的语句。

5.7.3 循环结构

当型循环结构可以使用while语句实现：

```
while (<表达式>)
    <循环体>
```

其中的＜循环体＞可以是一个语句，也可以是一个分程序：

```
while(<表达式>)
{
    ……
}
```

while语句的执行过程见图5-17。当表达式的结果不为0时反复执行其循环体内的语句或者分程序，直到表达式的值为0时退出循环。所以在设计当型循环时要注意在其循环体内应该有修改＜表达式＞的部分，以此确保在执行了一定次数之后可以退出循环，

否则循环永不结束，就成了“死循环”。

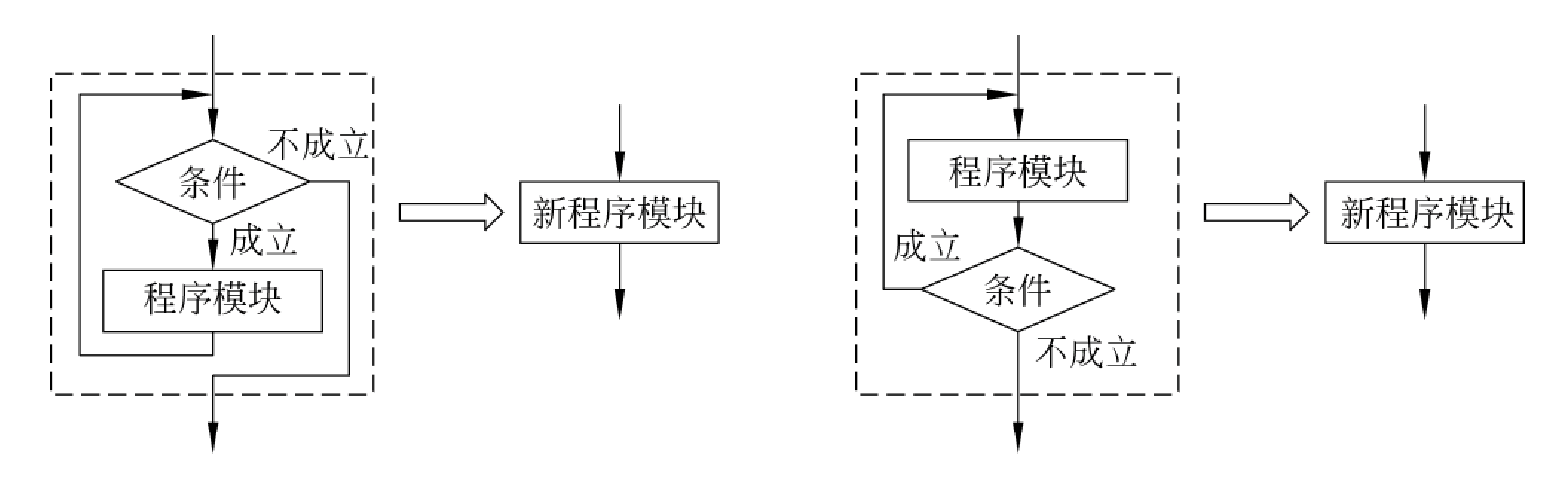


图 5-17　while 和 do-while 循环执行的过程

在图 5-17 中也可以看到，一个循环可以看成一个新的程序模块，而这个模块又可以看成组成其他循环的循环体。也就是说，循环是可以嵌套的。

直到型循环结构可以使用 do-while 语句实现：

```
do
{
    <循环体>
}while(<表达式>);
```

除此而外，C++ 还提供了一种使用起来更为方便灵活的 for 语句。其控制流程如图 5-18 所示。格式为

```
for(<表达式 1>;<表达式 2>;<表达式 3>)
    <循环体>
```

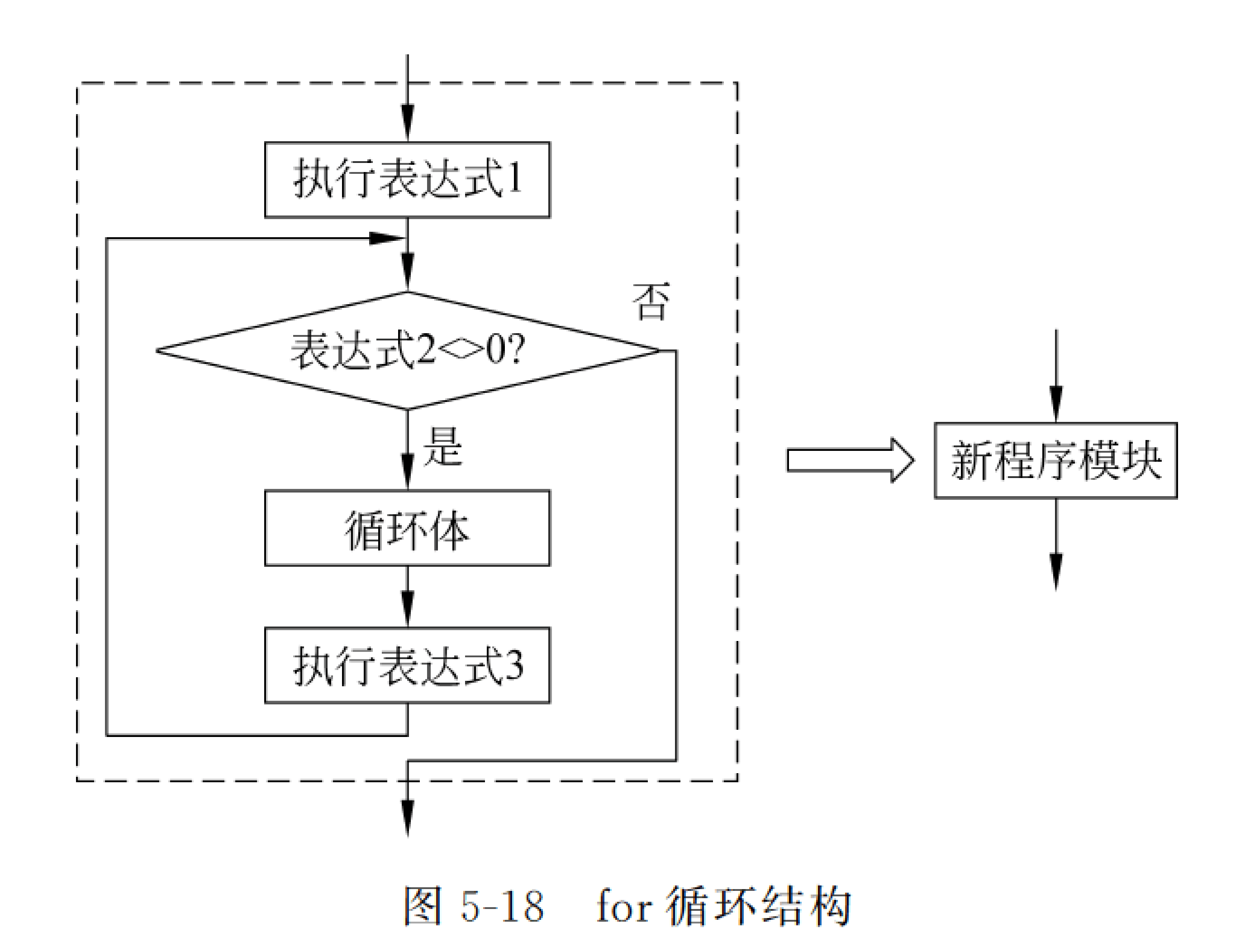


图 5-18　for 循环结构

和 while 语句的情况类似，for 语句的循环体既可以是一条语句，也可以是一个分程序。for 语句最常见的用途是构造指定重复次数的循环结构。例如：

```
for (i=0; i<10; i=i+1)
{
    ……
}
```

用于实现重复 10 次的循环。虽然用 while 语句和 do-while 语句也可以构造出这样的循环，但使用 for 语句更简单、直观。特别是在处理数组时，大多数程序员都喜欢使用 for 语句。

5.7.4 其他控制转移语句

C 语言提供的控制转移语句除了前面介绍的 if-else 语句、while 语句、do-while 语句和 for 语句以外，还有如下一些控制语句。

1. switch 语句

switch 语句用于实现多重分支，其格式为

```
switch(<整型表达式>)
{
    case<数值 1>:
        ……
    case<数值 2>:
        ……
    case<数值 3>:
        ……
    ⋮
    default:
        ……
}
```

其中 default 模块也可省略。switch 语句的执行过程是：首先计算整型表达式的值，然后将其结果与每一个 case 后面的数值常量依次进行比较，如果相等则执行该 case 模块中的语句，然后依次执行其后每一个 case 模块中的语句，无论整型表达式的值是否与这些 case 模块的进入值相同。如果需要在执行完本 case 模块以后就跳出 switch 语句，则可以在 case 模块的最后加上一个 break 语句，这样才能实现真正的多路选择。如果整型表达式的值与所有 case 模块的进入值无一相同，则执行 default 模块中的语句。带有 break 语句的 switch 多分支结构的框图如图 5-19 所示。

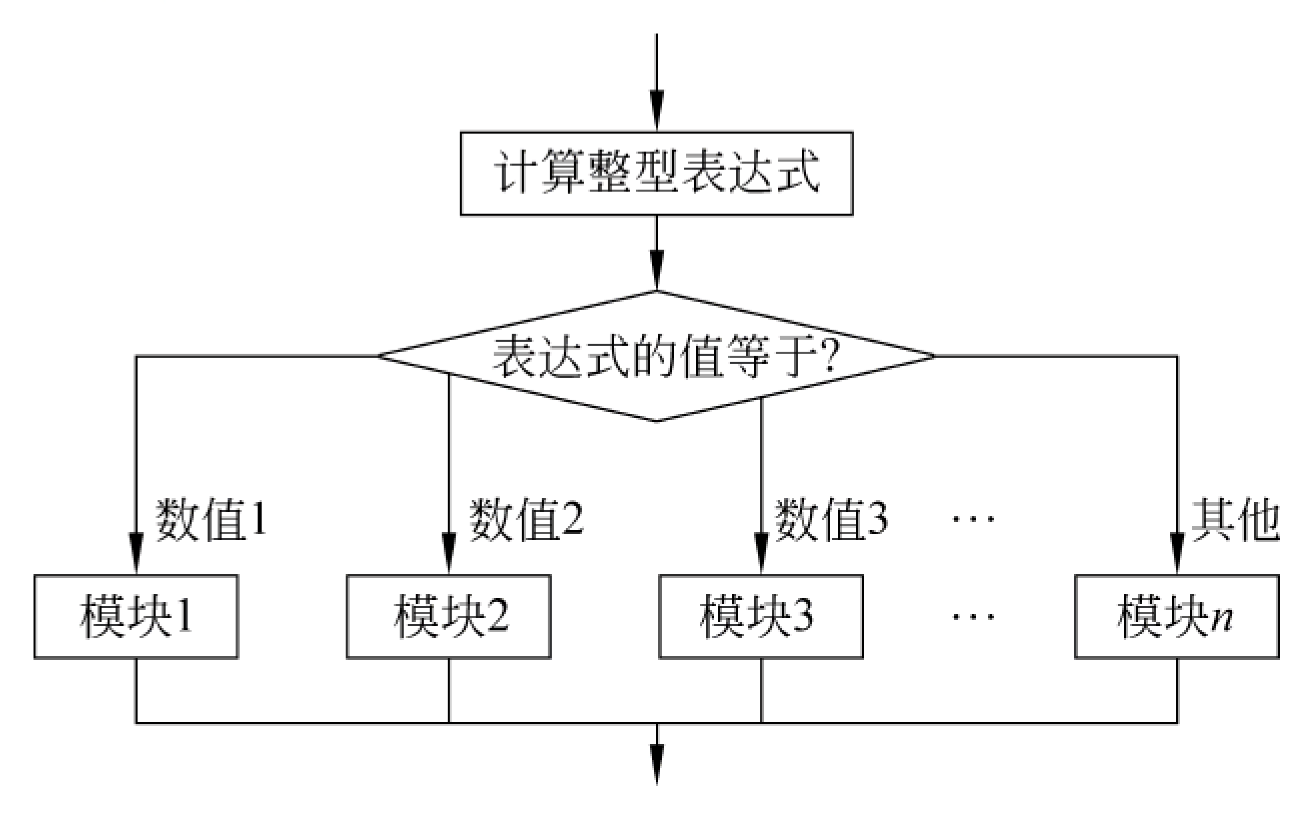


图 5-19 switch 语句(带 break 语句)

【例 5-20】 编写一个程序，将百分制的学生成绩转换为优秀、良好、中等、及格和不及格的 5 级制成绩。标准如下。

优秀：100～90 分；

良好：80～89 分；

中等：70～79 分；

及格：60～69 分；

不及格：60 分以下。

算法：使用 switch 语句构成的多分支结构编写这个程序。switch 语句根据具体的数值判断执行的路线，而现在的转换标准是根据分数范围，因此，构造一个整型表达式 old_grade/10 用于将分数段化为单个整数值。例如，对于分数段 60～69 中的各分数值，上述表达式的值均为 6。再配合以在 switch 语句的各 case 模块中灵活运用 break 语句，即可编写出所需的转换程序。

程序：

```
#include <stdio.h>
int main()
{
    int old_grade,new_grade;
    printf("Please input the score: ");
    scanf("%d",&old_grade);
    switch (old_grade/10)
    {
        case 10:
        case 9:
            new_grade=5;
            break;
        case 8:
            new_grade=4;
            break;
        case 7:
            new_grade=3;
            break;
        case 6:
            new_grade=2;
            break;
        default:
            new_grade=1;
    }
    printf("Before transfomed,the score is %d\n");
    printf("After transfomed,the score is %d\n");
    return 0;
}
```

输入和输出：

```
Please input the score: 85
Before transformed,the score is 85
After transformed,the score is 4
```

分析：该程序将用户输入的百分制的分数值(0～100)转换为5级制成绩：5代表优秀,4代表良好……1代表不及格。请注意,switch语句的第1个case模块中没有任何语句(包括break),因此进入该模块时(原成绩为100分)将直接转入第2个case模块(处理原成绩在90～99分之间)中继续执行。

2. goto语句和语句标号

C语言允许在语句前面放置一个标号,其一般格式为

```
<标号>:<语句>;
```

标号的取名规则和变量名相同,即由下划线、字母和数字组成,第一个字符必须是字母或下划线,例如：

```
ExitLoop: x=x+1;
End: return x;
```

在语句前面加上标号主要是为了使用goto语句。goto语句的格式为

```
goto <标号>;
```

其功能是改变语句执行顺序,转去执行前面有指定标号的语句,而不管其是否排在当前语句之后。C语言的goto语句只能在本函数模块内部进行转移,不能由一个函数中转移到另一个函数中。由于结构化程序设计方法主张尽量限制goto语句的使用,因此在这里不对goto语句做过多的讨论。

3. break语句和continue语句

break语句的格式为

```
break;
```

前面已经介绍过,将该语句用在switch语句中,可以使程序流程跳出switch结构。

如果将break语句用于循环语句,它可以使流程立即跳出包含该break语句的各种循环语句,即提前结束循环,接着执行循环下面的语句。在循环语句中使用的break语句一般应和if语句配合使用,例如：

```
while(<条件 1>)
{
    ……
    if(<条件 2>)
        break;
```

```
    ……
}
```

以上结构的框图如图 5-20 所示。

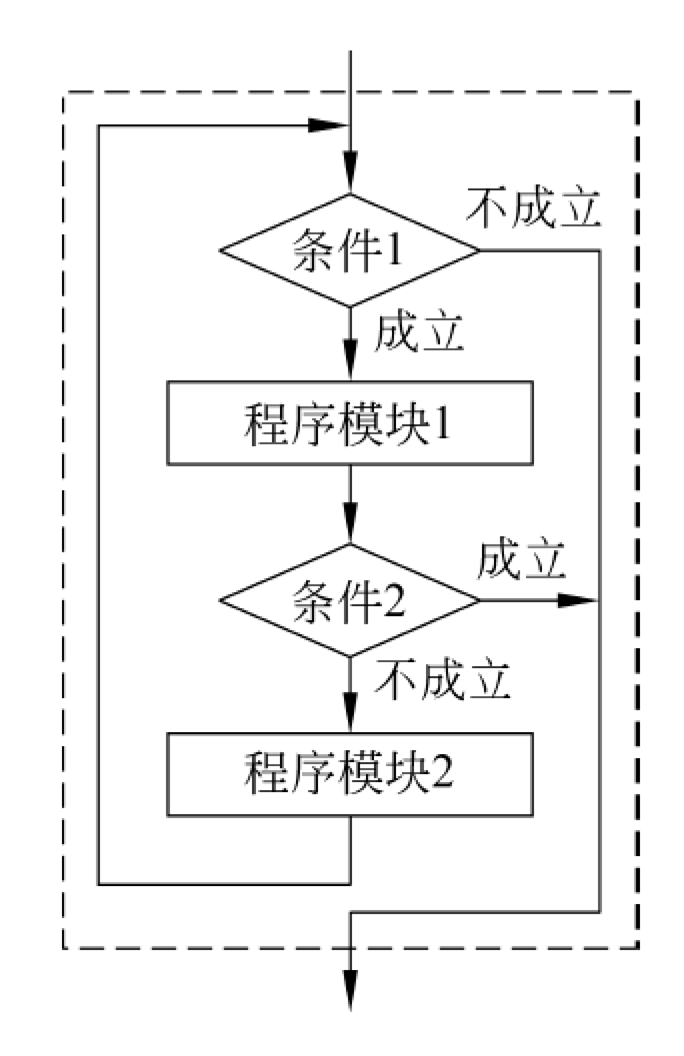
continue 语句用于提前结束本轮循环，即跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判断，可用于 while、do-while 和 for 语句中。其格式为

```
continue;
```

continue 语句的用法和 break 语句相似，均应和 if 语句配合使用。仍以 while 语句为例：

```
while(<条件 1>)
{
    ……
    if(<条件 2>)
        continue;
    ……
}
```

其执行框图如图 5-21 所示。





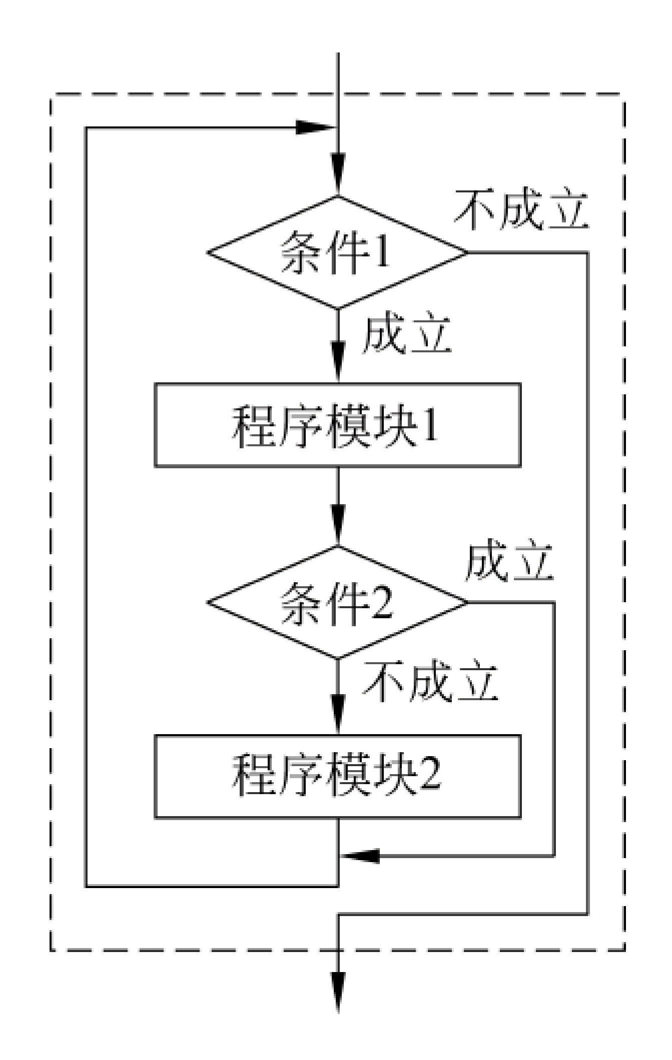


图 5-20　使用 break 语句的循环结构　　图 5-21　使用 continue 语句的循环结构

在循环中使用 break 语句和 continue 语句的区别是：break 语句是结束整个循环的执行，再不进行条件判断，而 continue 语句则只结束本次循环，而不终止整个循环过程。其实，break 语句和 continue 语句都是变相的 goto 语句。在某些应用问题的解决中恰当地使用这些语句，可以使程序的表达比较清晰，同时仍然满足结构化程序的基本特征：每个程序模块只有一个入口和一个出口，可以自上而下地阅读。

5.7.5　控制结构例题

【例 5-21】　计算 $e=1+\frac{1}{1!}+\frac{1}{2!}+\cdots+\frac{1}{n!}+\cdots$，当通项 $\frac{1}{n!}<10^{-7}$ 时停止计算。

算法：定义 3 个工作变量 e、n 和 u，分别用于存放已计算出的结果近似值、当前项序号和当前通项值，则伪代码算法为

```
e=1.0; n=1; u=1.0;
while(通项 u大于等于 10⁻⁷)
{
    计算新的通项值 u=u/n;
    将新通项值加到结果近似值上;
    准备处理下一项 n=n+1;
}
```

程序：

```c
#include <stdio.h>
int main()
{
    double e=1.0;
    double u=1.0;
    int n=1;
    while(u>=1.0e-7)
    {
        u=u/n;
        e=e+u;
        n=n+1;
    }
    printf("e=%lf(n=%d)\n",e,n);
    return 0;
}
```

输出：

```
e=2.71828(n=12)
```

分析：根据计算结果中打印出的项数 n，表明该级数收敛相当快，仅计算到前 12 项其截断误差便已小于 10^{-7}。

【例 5-22】 使用 do-while 结构重新编写例 5-21 的程序。

程序：

```c
#include <stdio.h>
int main()
{
    double e=1.0;
    double u=1.0;
    int n=1;
    do
    {
```

```
        u=u/n;
        e=e+u;
        n=n+1;
    }while(u>=1.0E-7);
    printf("e=%lf (n=%d)\n",e,n);
    return 0;
}
```

输出：

```
e=2.71828(n=12)
```

【例 5-23】 求水仙花数。如果一个三位数的个位数、十位数和百位数的立方和等于该数自身，则称该数为水仙花数。编写程序求出所有的水仙花数。

算法：对 100～999 的三位数的范围内所有的数一一进行检验，考察其是否符合水仙花数的定义：

```
for(n=100;n<=999;n=n+1)
    if(n是水仙花数)
        打印 n的分解形式;
```

程序：

```
#include<stdio.h>
int main()
{
    int n,i,j,k;
    for(n=100; n<=999; n=n+1)
    {
        i=n/100;                    //取出 n的百位数
        j=(n/10)%10;                //取数 n的十位数
        k=n%10;                     //取出 n的个位数
        if(n==i*i*i+j*j*j+k*k*k)
        printf("%d=%d^3+%d^3+%d^3\n",n,i,j,k);
    }
    return 0;
}
```

输出：

```
153=1^3+5^3+3^3
370=3^3+7^3+0^3
371=3^3+7^3+1^3
407=4^3+0^3+7^3
```

分析：在程序中利用了 C++ 的整数除法和求余运算从一个三位数中分离出其个位、十位和百位数。

【例 5-24】 猜幻数游戏。系统随机给出一个数字(即幻数),游戏者去猜,如果猜对,打印成功提示,否则打印出错提示,并提示游戏者下一次的猜测方向,最多可以猜 5 次。

算法:程序运用随机数产生函数 rand(),调用该函数可产生 0～32 767 的任意一个数。

```
for(i=0; n<=5; i=i+1)
    if(猜对)
        打印成功提示;
else
    打印出错提示;
```

程序:

```
#include <stdio.h>
int main()
{
    int magic;
    int guess;
    magic=rand();
    printf("Guess the magic number. It is between 0 and 32767.\n");
    for(int i=1; i<=5; i=i+1)
    {
        sacnf("%d",&guess);
        if(guess==magic)
        {
            printf("***Right***\n");
            break;
        }
        else
        {
            if(i==5)
                printf("The %d time is wrong. End of game!\n",i);
            else
            {
                if(guess<magic)
                    printf("You have been wrong for %d time(s). Please try a bigger one.\n",i);
                else
                    printf("You have been wrong for %d time(s). Please try a smaller one.\n",i);
            }
        }
    }
    return 0;
}
```

输入和输出：

```
Guess the magic number. It is between 0 and 32767.
30
You have been wrong for 1 time(s). Please try a bigger one.
1000
You have been wrong for 2 time(s). Please try a smaller one.
50
You have been wrong for 3 time(s). Please try a smaller one.
40
You have been wrong for 4 time(s). Please try a bigger one.
41
***Right***
```

【例 5-25】 输入一个整数，然后显示该整数的所有因子并统计因子的个数。

算法：要找出某个整数 n 的所有因子，可以用 $1\sim n$ 之间的每个整数去除 n，余数为零的即为因子，显然，每个整数可以通过循环实现，统计个数可以设置一个用于计数的变量 count 实现。

```
#include <stdio.h>
int main()
{
    int n,i,count=0;
    printf("Please input a integer\n");
    scanf("%d",&n);
    i=1;
    while(i<=n)
    {
        if(n%i==0)
        {
            printf("%d,"i);
            count++;
        }
        i++;
    }
    printf("\ncount=%d,count);
    return 0;
}
```

【例 5-26】 找出 1～10000 之间的所有同构数。

算法：一个正整数 m，如果是它的平方数的尾部，则称 m 为同构数。例如，6 是其平方数 36 的尾部，76 是其平方数 5776 的尾部，6 与 76 都是同构数。

在具体判断时，本例采用这样的方法：对 n 位的整数 m，取出其平方数 $m*m$ 右边的 n 位进行判断，方法是用 $m*m$ 除以 10 的 n 次方取余数。

程序：

```
#include <stdio.h>
int main()
{
    int i;
    printf("1~10000之间的所有同构数如下：\n");
    for(i=1;i<=10000;i++)
    {
        if(i<10 && (i*i)%10==i)
            printf("%d,%d\n",i,i*i);              //1位整数
        else
            if(i<100 && (i*i)%100==i)
                printf("%d,%d\n",i,i*i);          //2位整数
                else
                if(i<1000 && (i*i)%1000==i)
                    printf("%d,%d\n",i,i*i);      //3位整数
                else
                    if((i*i)%10000==i)
                        printf("%d,%d\n",i,i*i);  //4位整数
    }
}
```

5.8 应用示例

【例 5-27】 模拟仿真是计算机应用的一个极为重要的方面。通过计算机进行模拟试验，不仅可以节约大量的时间和费用，而且能提高实验数据的准确性和可靠性，甚至完成一些常规实验手段无法实现的实验研究，如核爆炸试验、天体试验、航天器飞行试验等。下面是一个简单的模拟仿真例子。

在码头酒馆和游船之间搭了一条长 20m、宽 4m 的跳板，醉酒的船员和游客回艇时必须通过这个跳板。通过跳板时，有 3 种可能的结果：

(1) 向前走，回到游船上休息，不再出来。

(2) 转身回到酒馆，重新开始喝酒，不再出来。

(3) 左右乱晃，落入水中淹死。

醉酒者每次走一步，一步走 1m，而且他们向前走的概率是 0.7，向左走、向右走和向后走的概率各为 0.1。现在假设开始时他们都是站在酒馆的门口，请编写程序模拟出若干个醉酒者的最终行为结果。

算法：为了模拟醉酒者的行为，需要有一个随机数产生函数，每产生一个数相当于醉酒者走了一步。C++ 提供了这样的一个函数，即 rand()，所以只需要直接使用就行了。

将醉酒者的行为代码化，用不同的整数来表示向前、后、左、右走。因为向各个方向走

的概率不同，分别是0.7、0.1、0.1和0.1，所以如果用0～9的整数来表示，可以假设0为向左，1为向右，2为向后，3～9为向前。

采用坐标将行走轨迹量化。将坐标原点取在跳板的中心，x轴从酒馆指向船的方向，跳板的两个邻水边的y坐标分别$y=2$和$y=-2$。这样，醉酒者开始所处的位置，即酒馆门口，x坐标为-10，回到船上x坐标为10。

程序：

```
//Example 模拟醉酒者行为程序
#include <stdio.h>
#include <math.h>
#define SHIP 1
#define BAR 2
#define WATER 3
//一个醉酒者行为的模拟仿真
int drunkard(void)
{
    int x=-10;                    //记录醉酒者的 x 坐标,开始时在酒馆门口
    int y=0;                      //记录醉酒者的 y 坐标,开始时在跳板的中央
    int step=0;                   //step 记录醉酒者一共走了多少步
    while(abs(x)<=10&&abs(y)<=2)
    {
        switch(rand()%10)
        {
            case 0:               //向左走
                y=y-1;
                break;
            case 1:               //向右走
                y=y+1;
                break;
            case 2:               //向后走
                x=x-1;
                break;
            case 3:               //向前走
            case 4:
            case 5:
            case 6:
            case 7:
            case 8:
            case 9:
            x=x+1;
        }
        step=step+1;
    }
    if(x<-10)
    {
```

```
            printf("After %d steps,the man returned to the bar and drunk again\n",step);
            return BAR;
        }
        else
        {
            if(x> 10)
            {
                printf("After %d steps,the man returned to the ship\n",step);
                return SHIP;
            }
            else
            {
                printf("After %d steps,the man dropped into the water\n",step);
                return WATER;
            }
        }
}
//反映若干个醉酒者最终行为的模拟仿真的主函数
int main()
{
        int drunkardnumber;             //醉酒者总数
        int shipnumber= 0;              //到达船上的人数
        int barnumber= 0;               //返回酒馆的人数
        int waternumber= 0;             //掉进水中的人数
        printf("Please input the number of drunkard\n");
        scanf("%d",&drunkardnumber);
        for(int i= 0; i< drunkardnumber; i= i+ 1)
        {
            switch(drunkard())
            {
                case SHIP:
                    shipnumber= shipnumber + 1;
                    break;
                case BAR:
                    barnumber= barnumber + 1;
                    break;
                case WATER:
                    waternumber= waternumber + 1;
                    break;
            }
        }
        printf("******************************\n");
        printf("Of all the %d drunkards:\n",drunkardnumber);
        printf("%d returned to the ship\n",shipnumber);
        printf("%d went to the bar and drunk again\n",barnumber);
        printf("%d dropped into the water\n",waternumber);
```

```
    return 0;
}
```

【例 5-28】 爬动的蠕虫。如图 5-22 所示，一条蠕虫，长度为 1 英寸，在一口深为 n 英寸的井的底部。已知，蠕虫每分钟可以向上爬 u 英寸，但必须休息 1 分钟才能接着往上爬。在休息的过程中，蠕虫又下滑了 d 英寸。就这样，上爬和下滑重复进行。请问，蠕虫需要多长时间才能爬出井？不足一分钟按一分钟计，并且假定只要在某次上爬过程中蠕虫的头部到达了井的顶部，那么蠕虫就完成任务了。初始时，蠕虫是趴在井底的（即高度为 0）。请编程模拟蠕虫爬动，求蠕虫爬出井的时间。

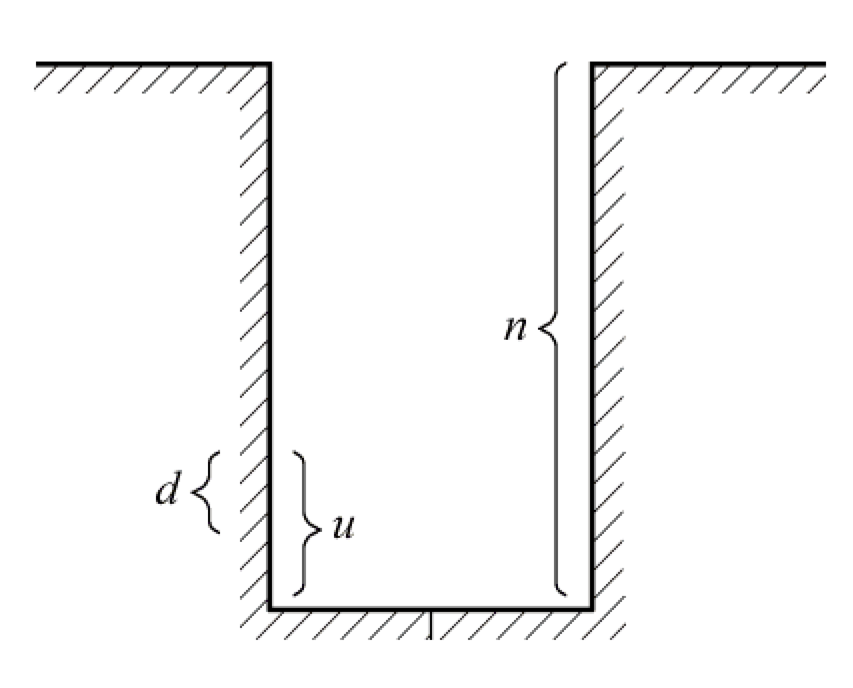


图 5-22　爬动的蠕虫

程序输入的测试数据占一行，为 3 个正整数：n,u,d，其中 n 是井的深度，u 是蠕虫每分钟上爬的距离，d 是蠕虫在休息的过程中下滑的距离。假定 $0<d<u, 0\leqslant n<100$。$n=0$ 表示输入数据结束。对输入的测试数据，输出一个整数，表示蠕虫爬出井所需的时间（分钟）。

问题分析：整个过程可以通过一个永真循环实现。在永真循环里，先是上爬一分钟，蠕虫的高度要加上 u，然后判断是否达到或超过了井的高度。如果是则退出循环；如果不是则要下滑 d 距离。也就是说，执行一次循环，实际上分别上爬了一分钟和下滑一分钟。是否退出循环是在上爬后判断的。循环结束条件就是爬出井，即蠕虫当前高度大于井的深度。

程序：

```
#include <stdio.h>
int main()
{
    int n,u,d;                      //井的深度、蠕虫每分钟上爬和下滑的距离
    int time,curh;                  //所需时间、蠕虫当前的高度
    while( 1 )
    {
        scanf( "%d%d%d",&n,&u,&d );
        if(!n) break;
        curh=0,time=0;              //当前高度及所花时间
        while(1)
        {
            curh +=u;               //每爬一次,上升 u距离
            time++ ;
            if(curh>=n ) break;
            curh -=d ;              //休息时滑下 d距离
            time++ ;
        }
        printf( "%d\n",time );
```

```
    }
    return 0;
}
```

习　　题

1. 编写一个程序，要求完成以下要求：
   (1) 提示用户输入任意的 3 个小数。
   (2) 显示这 3 个小数。
   (3) 将这 3 个小数相加，并显示其结果。
   (4) 将结果按四舍五入方法转换成整数并显示。
2. 为例 5-13 添加数据检验部分。给出三边长，检验其是否能构成一个三角形的方法是检查是否任意两边和均大于第三边。如果检验不合格，输出信息“Error Data!”
3. 从键盘输入任意 3 个整数，然后输出这 3 个数并计算其平均值。
4. 编写一个程序，将字符串 Love 译成密码，译码方法采用替换加密法，其加密规则是：将原来的字母用字母表中其后面的第 3 个字母的来替换，如字母 c 就用 f 来替换，字母 y 用 b 来替换。提示：分别用 4 个字符变量来存储'L'、'o'、'v'和'e'，利用 ASCII 表中字母的排列关系，按照译码方法对各个变量进行运算后输出即可。
5. 输入一个总的秒数，将该秒数换算为相应的时、分、秒。如输入 3600 秒，则输出结果为 1 小时，输入 3610 秒，则结果为 1 小时 10 秒，通过除法和求余运算完成。
6. 编写程序，定义两个整数，用户通过键盘输入两个整数，程序计算它们的和、差、积、商并输出。
7. 编写计算阶乘 $n!$ 的程序。
8. 编写程序求斐波那契数列的第 n 项和前 n 项之和。斐波那契数列是
   0,1,1,2,3,5,8,13,…
   其通项为
   $F_0=0$；
   $F_1=1$；
   $F_n=F_{n-1}+F_{n-2}$
9. 编程求 $\arcsin x\approx x+\frac{1}{2\times 3}x^2+\frac{1\times 3}{2\times 4\times 5}x^5+\cdots+\frac{(2n)!}{2^{2n}(n!)^2(2n+1)}x^{2n+1}+\cdots$，其中 $|x|<1$。
   提示：结束条件可用 $|u|<\varepsilon$，其中 u 为通项，ε 一般可以取 10^{-7}。
10. 求解猴子吃桃问题。猴子在第一天摘下若干个桃子，当即就吃了一半，又感觉不过瘾，于是就多吃了一个。以后每天如此，到第 10 天时，就只剩下了一个桃子。请编程计算第一天猴子摘的桃子个数。
11. 所谓孪生素数是指间隔为 2 的相邻素数，例如最小的孪生素数是 3 和 5，5 和 7 也是。

找出 2～200 之间的孪生素数。

12. 从键盘输入一个正整数，然后将该整数分解为 1 和各个质因子相乘，如果输入的整数本身就是质数，则应分解为 1 和该数本身相乘。

13. 某地发生了一起犯罪案件，警察经过审问，做出了以下判断：

(1) A、B 至少有 1 人作案。

(2) A、E、F 中至少有 2 人参与作案。

(3) A、D 不可能都是案犯。

(4) B、C 或同时作案，或与本案无关。

(5) C、D 中有且仅有 1 人作案。

(6) 如果 D 没有参与作案，那么 E 也不可能参与作案。

请利用学过的关于逻辑运算和流程控制的方法，设计解答方案，并编程输出所有的案犯。

14. 使用循环嵌套的结构找出 100 以内的勾股数，要求找出 3 个数 a、b、c，它们满足以下的条件：

$$a^2 + b^2 = c^2$$

$$a < b < c$$

15. 在屏幕上输入多个正整数，将输入的正整数累加，直到输入为负数或 0 时，停止读取数据，计算读取的正整数的和以及平均数，要求使用 while/do-while 循环结构和 break 语句实现（这个程序不用 break 语句是可以实现的，但比较烦琐）。

16. 编写一个程序，寻找用户输入的几个整数中的最小值，并假定用户输入的第一个数值指定后面要输入的数值个数。例如，当用户输入数列为 5 20 15 300 9 700 时，程序应该能够找到最小数 9。

17. 有一个分数序列

$$\frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}, \frac{21}{13}, \cdots$$

即后一项的分母为前一项的分子，后项的分子为前一项分子与分母之和。编写程序，求其前 n 项之和。

18. 编写程序，求 $a + aa + aaa + aaaa + \cdots + aa\cdots a$（$n$ 个），其中 a 为 1～9 之间的整数。例如：

当 $a=1$，$n=3$ 时，求 1＋11＋111 之和。

当 $a=5$，$n=7$ 时，求 5＋55＋555＋5555＋55555＋5555555 之和。

19. 一个采购员去银行兑换一张 d 元 c 分的支票，结果出纳员错给了 c 元 d 分。采购员用去了 23 分之后才发觉有错，于是清点了余额尚有 $2d$ 元 $2c$ 分。编程求解该支票面额。

20. 编写程序，输入 3 个整数，求这 3 个整数的最大公约数和最小公倍数。

第 6 章 数组、函数和指针

引言

第 5 章学习了一些基础的 C 语言语法知识。在本章中，将继续学习数组、函数和指针方面的内容。基本的数据类型难以描述复杂的现实世界，C 语言提供了数组和结构体等更为丰富的数据类型来描述复杂数据。结构化程序设计（structured programming）是一种编程典范。它采用子程序、程序码区块（block）、for 循环以及 while 循环等结构来取代传统的 goto 语句。希望借此来改善计算机程序的明晰性、质量以及开发时间，并且避免写出面条式代码（spaghetti code）。函数则是结构化编程的基础。而 C 语言功能的强大以及自由性，很大部分体现在其灵活的指针运用上。因此，说指针是 C 语言的灵魂一点都不为过。

教学目的

- 掌握数组的使用。
- 掌握结构体和共用体的使用。
- 会使用枚举。
- 掌握 C 语言函数编写与使用。
- 掌握指针的使用。
- 具有初步的结构化程序设计能力。

6.1 数 组

第 5 章中学习了一些基本的数据类型，这些变量和常数多用来表示少量相互之间没有多少内在联系的数据，或表示一个单独的数据项。而在实际应用中，只用几个变量的情况是极少的，更多的情况是处理大批量相同类型或不同类型的数据。大量的成批数据则需要使用更为复杂的数据结构来存放，这时一般都会使用数组。

所谓数组是一组相同类型的变量，用一个数组名标识，其中每个变量（称为数组元素）

通过该变量在数组中的相对位置(称为下标)来引用。数组可以是一维的,也可以是二维或者更高维的。二维以上数组统称为多维数组。

和变量一样,数组也遵循"先定义,后使用"的原则。定义数组时,系统为数组中的每个元素分配相同大小的存储单元,而整个数组在内存中分配连续多个存储单元。图 6-1 分别给出了一维、二维和三维数组中的数组元素排列方法。

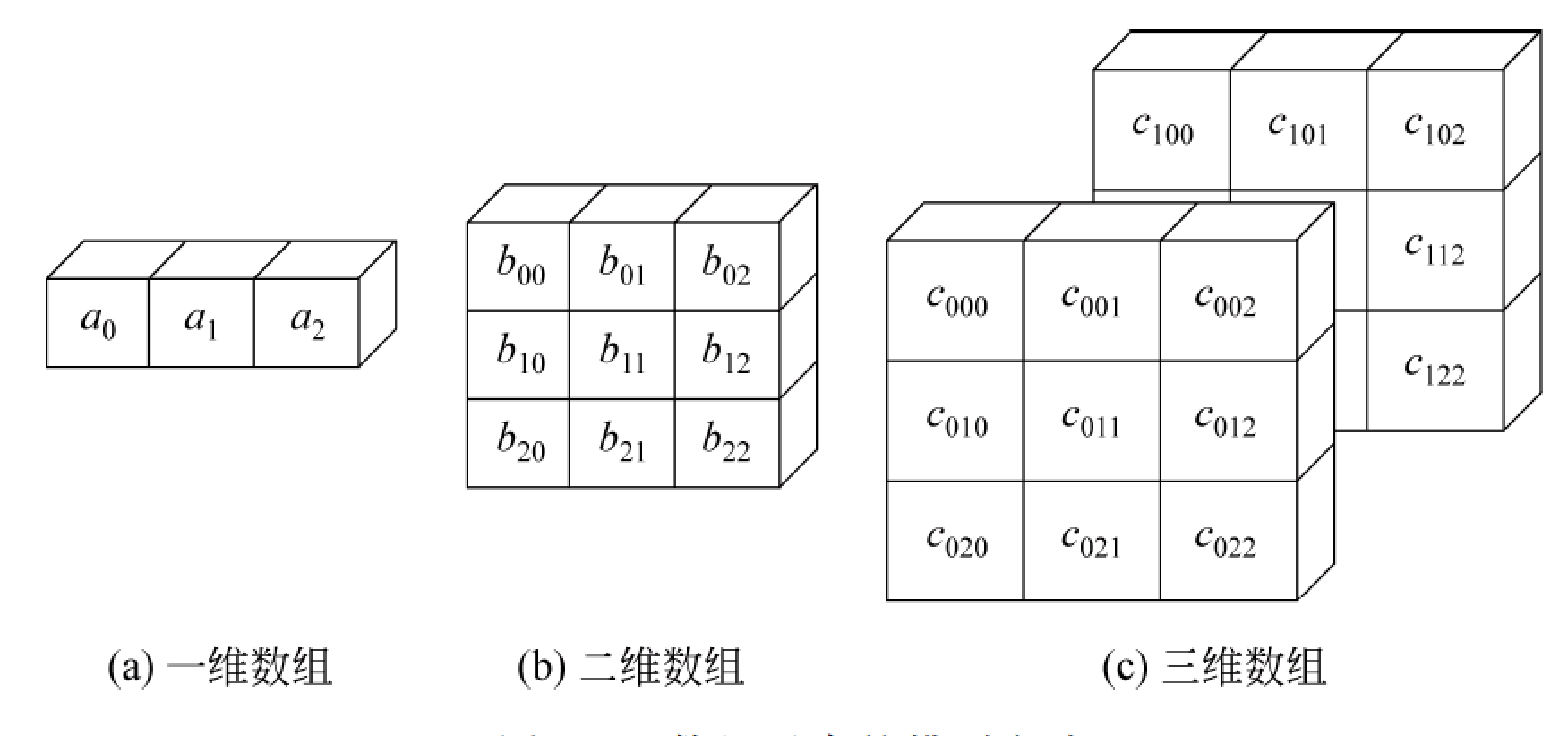


图 6-1　数组元素的排列方式

图 6-1 的这种排列方法仅仅是数组的逻辑结构,逻辑结构是从逻辑关系(某种顺序)上观察数据,它独立于计算机,可在理论上、形式上进行研究、运算。而要研究数组的所有数组元素是如何在存储器中占用一片连续的存储单元的情况,就要涉及数组的物理结构,物理结构也称存储结构,是逻辑结构在计算机中的实现,它依赖于计算机。

6.1.1　一维数组

一维数组用于存放一行或一列数据,数组要占用一定的内存空间。要为数组分配存储空间,就必须先要对数组进行定义,数组的定义方法与变量相同,只是要在数组名后面加上用方括号括起来的各维长度即可。

一维数组说明语句格式如下:

<类型><数组名>[<常量表达式>];

其中,<数组名>的命名规则同变量名,<常量表达式>必须用方括号括起来,其值给出数组元素的个数,<类型>(如 int、char、double 等)指出数组中元素的数据类型。例如:

```
int array[10];              //说明了一个有 10 个元素的整型数组
```

要注意的是:数组元素的下标从 0 开始编号。例如,array[0]是数组 array 中的第一个数组元素。上述说明语句说明了一个有 10 个元素的整型数组,其数组名为 array,每个元素为整型数据。各元素通过不同的下标来区分,分别为 array[0],array[1],array[2],…,array[9]。同时系统也为该数组分配了 10 个连续的存储空间,如图 6-2 所示。

由此可见,一维数组的逻辑结构是由一串数据构成的向量,每个元素的下标值确定了各元素在此数据表中的位置,其物理存储结构和逻辑结构是一样的。

array[0]	array[1]	array[2]	array[3]	array[4]	array[5]	array[6]	array[7]	array[8]	array[9]

图 6-2　一维数组在内存中的排列方式

在声明数组的同时也可以对其初始化，一般形式为

<类型><数组名>[<常量表达式>]={<常量 1>,<常量 2>,…};

如果在声明数组时给数组的每一个元素都提供初值，就可以不必指定数组大小。这时数组中元素的个数就是初始化值列表中元素的个数，例如：

```
double x[5]={1.2,3.2,-3.5,6.6,-4.1};
```

等价于

```
double x[]={1.2,3.2,-3.5,6.6,-4.1};
```

数组的使用和一般变量不同，C 语言不允许对一个数组进行聚集操作，即不能将整个数组作为一个单元操作。例如，假设数组 a 和 b 是相同类型和大小的数组，如果想将数组 a 的值赋给 b，下面的语句是错误的：

```
b=a;                    //不合法的语句
```

要想实现这个功能，就必须进行对应元素的赋值，一次只能给一个元素赋值。

同样，为数组输入输出数据、查找最大最小元素等操作也都不能以数组整体为对象，而是需要对数组进行遍历，最常用的处理方法是通过循环处理数组中的元素。例如：

```
//将数组中的所有元素置零
for(int i=0; i<N; i=i+1)
    array1[i]=0;
```

【例 6-1】　给一维数组输入 7 个整数，找出数组中的最大数。

算法：找数组中的最大元素这类问题可以利用扫描法解决。即以数组的第一个元素为基准，向后比较，如果遇到有比基准元素更大的元素，则将基准元素替换为该元素，直到数组中所有的元素均被扫描。这时得到的最新的基准元素就是数组中最大的元素。

程序：

```
#include <stdio.h>
int main()
{
    int array[7];
    printf("Please input an array with seven elements: \n");
    for(int i=0;i<7;i++)
        scanf("%d",&array[i]);
    int big=array[0];
    for(int j=0;j<7;j=j+1)
```

```
        if(array[j]>big)
            big=array[j];
    printf("max=%d\n",big);
    return 0;
}
```

输入：

```
2 1 7 3 12 4 9
```

输出：

```
max=12
```

6.1.2　二维数组

二维数组用于存放排列成行、列结构的表格数据，即矩阵形式的数据。定义二维数组时，除了给出数组名和数组元数的类型外，同时应给出二维数组的行数和列数。

定义格式如下：

<类型><数组名>[<常量表达式 1>][<常量表达式 2>]

例如：

```
int matrix[3][4];            //说明了一个 3行 4列的整型矩阵
```

与一维数组相似，二维数组同样定义了类型相同的一组变量，这些变量也称为数组元素或下标变量，行、列下标值也是从 0 开始，依次加 1。matrix[0][0]是矩阵 matrix 中的第 1 行第 1 列元素，位于矩阵的左上角。

二维数组的逻辑结构恰似一张表格，如数组 matrix 的逻辑结构排列顺序如图 6-3 所示。

matrix[0][0]	matrix[0][1]	matrix[0][2]	matrix[0][3]
matrix[1][0]	matrix[1][1]	matrix[1][2]	matrix[1][3]
matrix[2][0]	matrix[2][1]	matrix[2][2]	matrix[2][3]

图 6-3　二维数组的逻辑结构

二维数组的物理存储结构是以行次序优先进行内存分配，即先为第 1 行各元素分配存储单元，接着是第 2 行，第 3 行……每一行中的各个元素按列号递增次序进行分配，如图 6-4 所示。整个数组在内存中占据连续的一片存储单元，如数组 matrix 的物理存储结构。

和一维数组一样，二维数组的初始化也可以在定义时进行，有以下两种方法：

(1) 按照二维数组元素的物理存储次序给所有数组元素提供数据值。例如：

```
int matrix[3][4]={85,87,93,88,86,90,95,89,78,91,82,95};
```

（2）以行结构方式提供各元素数据值。

用花括号按行分组，为二维数组提供初值。例如，matrix 数组也可用下面的形式表示：

```
int matrix [3][4]= {{85,87,93,88},
                    {86,90,95,89},
                    {78,91,82,95}};
```

另外，允许在为二维数组初始化时省略行下标值，但列下标值不能省略。因此，下面的定义也是正确的：

```
int matrix [][4]= {85,87,93,88,86,90,95,89,78,91,82,95};
```

对于二维数组，通常使用二重循环结构控制其行列下标访问数组中的每一个元素。

例如：

```
//将矩阵 matrix 置成单位阵
for(int i= 0;i < 20;i= i+ 1)
{
    for(int j= 0;j < 20; j= j+ 1)
        matrix[i][j]= 0.0;
    matrix[i][i]= 1.0;
}
```

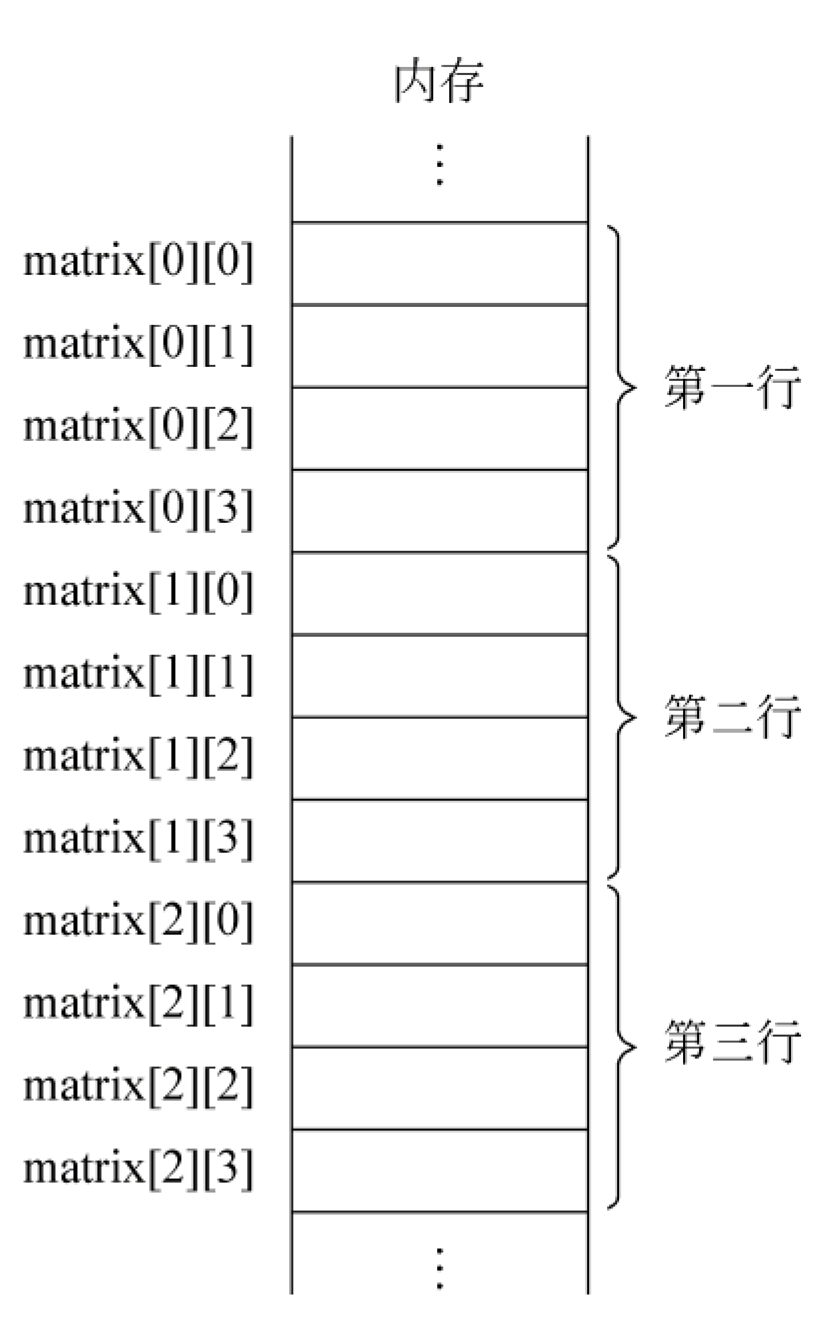


图 6-4　二维数组内存分配示意图

6.1.3　多维数组

C 语言编译器支持至少 12 个数组下标。

定义多维数组的一般形式是

<类型><数组名>[<常量表达式 1>][<常量表达式 2>]…[<常量表达式 n>]

例如：

```
float tri[2][3][3]          //说明了一个 2页 3行 3列的三维浮点型数组
```

三维数组的逻辑结构可以看成是若干张表格（或矩阵）的组合，其物理存储结构按自然顺序（即从下标序列对应的值从小到大顺序）在一片连续的内存中分配存储单元，如 tri 数组在内存中就以如下顺序存在：

tri[0][0][0]，tri[0][0][1]，tri[0][0][2]，tri[0][1][0]，tri[0][1][1]，tri[0][1][2]，tri[0][2][0]，tri[0][2][1]，tri[0][2][2]，tri[1][0][0]，tri[1][0][1]，tri[1][0][2]，tri[1][1][0]，tri[1][1][1]，tri[1][1][2]，tri[1][2][0]，tri[1][2][1]，tri[1][2][2]

多维数组的使用和一维数组或二维数组的用法基本类似，也是使用数组的元素，而不是数组名。访问多维数组元素常用的语法是

<数组名>[<常量表达式 1>][<常量表达式 2>]…[<常量表达式 n>]

可以用循环来处理多维数组。在使用数组元素时，要注意数组下标值应该在已定义的数组大小范围内，否则会出现错误的结果。

实际上，多于三维的数组并不常用。

6.2 字符型数组和字符串处理库函数

6.2.1 字符型数组的定义和初始化

C 语言使用字符型数组存放字符串数据并实现有关字符串的操作。由第 5 章字符串常数的存储格式可知，字符串包括一个结束符'\0'，所以在计算用于存放字符串的数组的大小时要考虑到这一点。例如，如果要设计一个能够存放最大长度为 80 个字符的字符串的数组，其长度应为 81。

字符型数组实际是数组元素为 char 类型的数组，其用法和普通数组相同。例如，设计一个字符型数组 weekday 用于存放星期的名称，并将字符串"MONDAY"存入其中，可以这样设计：

```
char weekday[7];
weekday[0]='M';
weekday[1]='O';
weekday[2]='N';
weekday[3]='D';
weekday[4]='A';
weekday[5]='Y';
weekday[6]='\0';
```

其中最后一句也可以直接写成

```
weekday[6]=0;
```

这个过程也是数组初始化的过程，除此以外，还可以采用其他方式：

```
char weekday [7]={'M','O','N','D','A','Y','\0'};
```

等价于下面的语句：

```
char weekday [7]={"MONDAY"};
char weekday [7]="MONDAY";
```

初始化后，weekday 数组的存储情况如图 6-5 所示。

weekday[0]	weekday[1]	weekday[2]	weekday[3]	weekday[4]	weekday[5]	weekday[6]
M	O	N	D	A	Y	\0

图 6-5 weekday 数组初始化后在内存中的存储情况

6.2.2 字符串的输入与输出

由前面知道，字符型数组的用法和普通数组基本相同。而和普通数组不同的是，字符型数组允许聚集操作，而普通数组的操作都只能通过循环对逐个元素完成。例如：

```
char weekday[7];
sacnf("%s",weekday);        //将从键盘输入的字符串存入字符型数组 weekday 中
```

值得注意的是，在上面的例子中，由于声明的字符型数组维长为 7，只能存储不超过 6 个字符。如果用户输入的字符串长度大于 6，系统会将输入的字符串顺序放在 weekday 后续的内存单元中，从而会使后续的内存单元中的数据被破坏，造成严重错误。

在字符型数组输入问题上另外一个要注意的是，scanf 一旦遇到空白字符会停止读入数据到当前变量中。例如：

```
char name[20];
scanf("%s",name);
```

当输入姓名"Cong Zhen"时，变量 name 中的字符串只有"Cong"。

由此可见，包含空格的字符串无法使用 scanf 来输入。解决办法是使用 gets 函数。gets 函数有 1 个参数，是字符数组变量。如语句

```
gets(name);
```

就将从键盘输入的字符串存入到字符型数组变量，而不管是否中间出现空格符。字符串的输出也可以直接对字符型数组进行操作，例如：

```
printf(weekday);            //将字符型数组 weekday 中的内容输出到屏幕上
```

【例 6-2】 字符串的输入与输出。

```
#include <stdio.h>
int main()
{
    char name1[20],name2[20];
    printf("Please input a name with blank(within 19 characters): \n");
    gets(name1);
    printf("Please input the name again\n");
    scanf("%s",name2);
    printf("Using function gets,the name string in the variable is: %s\n",name1);
    printf("Using function scanf,the name string in the variable is: %s\n",name2);
    return 0;
}
```

输入和输出：

```
Please input a name with blank(within 19 characters):
```

```
Cui Shuning
Please input the name again
Cui Shuning
Using function get,the name string in the variable is: Cui Shuning
Using function scanf,the name string in the variable is: Cui
```

6.2.3 字符串处理库函数

C 语言提供了一组用于字符串处理的库函数,可以完成许多常用的字符串操作。

strcpy():字符串复制。

strcat():字符串连接。

strchr():在字符串中查找字符。

strcmp():字符串比较。

strlen():求字符串长度。

strlwr():将字符串中的大写字母转换为小写字母。

strrev():反转字符串。

strstr():在字符串中查找另一个字符串。

strupr():将字符串中的小写字母转换为大写字母。

……

因为这些库函数的说明存放在头文件 string.h 中,所以如果要在程序中调用这些函数,还应该在源程序的最前面加上一个文件包含的编译预处理命令:

```
#include <string.h>
```

下面介绍其中几个常用的字符串操作函数的用法。

1. 求字符串的长度

```
int strlen(char * s);
```

其中的参数说明 char *s 是说明 s 是一个指向字符类型的指针。指针类型在第 7 章中介绍,目前只需知道该参数既可以使用字符串常数,也可以使用字符型数组就可以了。该函数的返回值为字符串中字符的个数(不包括字符串结束符)。例如,语句

```
len=strlen("This is a sample.");
```

执行后,变量 len 会被赋值 17。

2. 复制字符串

```
strcpy(char * destin,char * source);
```

该函数的功能为将字符串 source 的内容复制到字符型数组 destin 中。例如:

```
char weekday[11];
```

```
strcpy(weekday,"MONDAY");
```

注意,destin 的长度一定要比字符串 source 的实际长度大,否则会引起严重的运行错误。

3. 连接字符串

```
strcat(char * destin,char * source);
```

该函数的功能为将字符串 source 的内容复制到字符型数组 destin 中原来的字符串的后面,使两个字符串合并成一个字符串。使用该函数时特别要注意保证字符型数组 destin 的长度一定能够放得下合并后的整个字符串(包括最后的字符串结束符),否则也会引起严重的运行错误。

4. 字符串比较

```
int strcmp(char * string1,char * string2);
```

该函数的功能为比较两个字符串。比较是按字典序进行的,即在字典中排在前面的单词小于排在其后的单词。当然,一般的字符串中不但可以有英文字母,还可能有其他符号,这时各个符号之间的比较按 ASCII 码的顺序进行。

如果字符串 string1 小于字符串 string2,该函数返回一个负整数值;如果字符串 string1 等于字符串 string2,该函数返回 0;如果字符串 string1 大于字符串 string2,该函数返回一个正整数值。例如:

```
if(strcmp(weekday,"SUNDAY")==0)
    printf("Today we have a party.\n");
```

5. 大小写字母转换

```
strlwr(char * string);          //大写变小写
strupr(char * string);          //小写变大写
```

这两个函数的功能类似,都是转换字符串中的英文字母大小写,对字符串中的其他符号没有影响。例如:

```
strlwr(weekday);
```

如果转换前字符型数组中存放着字符串"MONDAY",则转换后其内容变为"monday"。其实,这些标准库函数并不神秘,我们自己也完全可以编写出同样功能的程序。

【例 6-3】 编写一个用来计算字符串长度的函数 mystrlen(),并用主函数验证。

算法:使用循环结构来控制字符个数的统计,循环结束条件是遇到结束标志符'\0'

程序:

```
#include <stdio.h>
//计算字符串的长度的函数
```

```
int mystrlen(char string[])
{
    int len=0;
    while(string[len]!='\0')
        len=len+1;
    return len;
}
//测试计算字符串长度的主函数
int main()
{
    char string[100];
    printf("Please input a string (within 99 characters): \n");
    gets(string);
    printf("The length of the string is: %d\n",mystrlen(string));
    return 0;
}
```

输入：

```
china
```

输出：

```
The length of the string is: 5
```

分析：该函数的构造非常简单。值得注意的有两点：一是作为参数的一维数组可以不写明数组元素的个数。但C语言规定，二维以上的数组，除了第一维以外均应注明维长。例如：

```
void set_empty(double matrix[][10])
```

另外，要注意区别字符串的长度和存放字符串的字符型数组的长度这两个不同的概念。字符串以结束符'\0'表示字符串的结束，字符串的长度是到字符串结束符之前的字符个数(不包括字符串结束符)。因此，字符串的长度要小于字符型数组的大小。

6.3 结构体类型

在设计数据处理方面的应用程序时，常会发现要处理的数据相当复杂。

以企业中常用的工资管理程序为例，它所需要处理的工资单数据就很庞杂，在每个员工的工资单上有姓名、部门、基本工资、岗位津贴、独生子女费、水电费、房租等项目，而在计算这些项目时还可能要用到职称/职务/工种、参加工作时间(工龄)、是否有独生子女、住房面积等数据，有时甚至要用到员工配偶及其子女的有关数据，如是否双员工、独生子女的出生日期以及是否在子弟学校和幼儿园上学等。这些数据的类型各异：有的可以用

整型数据表示;有的只能用浮点类型数据表示;有的是文字数据,要用字符串表示;还有些数据比较复杂,如日期,本身又有其内部结构。

为清晰起见,可以用层次结构表述如下。

【例 6-4】 工资单数据的层次结构。

01 工资单
  02 工作部门:字符串,最大长度为 10 个字符
  02 姓名:字符串,最大长度为 8 个字符
  02 职务(含职称、工种):代码,0～99
  02 参加工作时间
    03 年份:1900～2050
    03 月份:1～12
    03 日:1～31
  02 家庭情况
    03 婚否:0-否,1-是
    03 是否双员工:0-否,1-是
    03 独生子女出生日期,如无独生子女则填 1900.01.01
      04 年份:1900～2050
      04 月份:1～12
      04 日:1～31
    03 入托子女数:0～10
    03 住房面积:0～1000
  02 基本工资:0～10 000,保留两位小数
  02 岗位津贴:0～10 000,保留两位小数
  02 劳保福利:0～1000,保留两位小数
  02 独生子女费:0～10,保留两位小数
  02 房租:0～10 000,保留两位小数
  02 电费:0～10 000,保留两位小数
  02 水费:0～10 000,保留两位小数
  02 取暖费:0～1000,保留两位小数
  02 保育费:0～1000,保留两位小数
  02 实发工资:0～10 000,保留两位小数

分析:本例中采用了缩排方式表示上述工资单数据的层次结构。每个数据项前面有一个层次号,用以明确数据项之间的隶属关系。数据项名称后面可以填写该数据项的类型、数据范围以及其他注意事项。

面对这样复杂的数据结构,在编程时会遇到什么问题呢?

首先,前面介绍的几种简单数据类型无法表示这类复杂数据的内在联系;其次,由于各数据项的类型互不相同,“一个单位的工资单”无法用一个数组存放,只能对各数据项分别建立数组,这些数组中的数据颇难保持一致;最后,在程序中设置了过多的变量和数组,

而设置数组时又只能按类型一致的原则进行，不得不违反数据结构的内在联系。因此数据结构的复杂化带来了程序结构的复杂化，使程序难于设计，可读性降低，调试困难。

造成以上问题的原因就在于缺乏一种能够有效地表示复杂数据之间的内在联系的数据结构。

C 语言中的结构体类型就适用于说明这种具有层次结构的复杂数据。

结构体类型与前面介绍的简单数据类型的区别是，结构体类型本身的构造可以根据数据的具体情况进行设置，这一过程又称为定义结构体数据类型；在定义了结构体类型之后，即可用其声明该结构体类型的变量和数组，一旦声明之后，就可以使用这些变量和数组了。

一个结构体类型的变量可以用来表示一个数据处理对象包含的所有数据，与简单类型的变量一样可以作为函数的参数或返回值，也可用其构造结构体类型的数组，从而克服了使用简单数据类型编写复杂的数据处理类应用程序所出现的各种困难。

6.3.1 结构体类型的定义

结构体类型的定义方法如下：

```
struct <结构体类型名>
{
    <结构体类型的成员变量说明语句表>
};
```

例如，可定义一个表示日期的结构体类型：

```
struct Date
{
    int da_year;
    int da_mon;
    int da_day;
};
```

即一个日期类型的变量有 3 个成员变量：年份(da_year)、月份(da_mon)和日(da_day)。

在定义好结构体类型以后，就可以声明该类型的变量了，结构体变量的声明方法和其他类型的变量一样，例如，变量说明语句

```
struct Date yesterday,today,tomorrow;
```

就说明了 3 个日期类型的变量：yesterday、today 和 tomorrow。

6.3.2 结构体类型变量的使用

对结构体类型变量的成员变量的引用方法为

```
<结构体类型变量名>.<成员变量名>
```

例如：

```
today.da_year=2004;
today.da_mon=1;
today.da_day=22;
```

两个相同类型的结构体变量之间可以互相赋值。但和数组一样，不能将结构体变量作为一个整体输入输出，只能以结构体的成员作为基本变量，一次输入或输出结构变量中的一个成员。例如，下面的语句将输出结构体变量 today 的内容：

```
printf("%d年%d月%d日\n",today.da_year,today.da_mon,today.da_day);
```

6.3.3 数组和结构体

结构体的成员可以是数组，其使用方法和简单变量相同。例如：

```
struct StudentType
{
    char id[10];            //学号
    double score[5];        //五门课程成绩
    double GPA;             //平均分
};
struct StudentType xjtuStudent;
```

如果要访问结构变量 xjtuStudent 的第二门课程成绩，可以用如下方法：

```
xjtuStudent.score[1]
```

前面指出，数组是由一组相同类型的变量组成的，这些变量可以是简单变量，例如 int 或 double，也可以是复合类型，如结构体。

例如，在 5.1.1 节的学生例子中，如果某个班有 30 名学生，需要对这个班所有的学生的成绩情况进行处理，在定义完上面的数据结构后，因为这些学生的数据类型都是相同的，所以可以用一个有 30 个元素的数组来处理学生的数据，即进行如下变量声明：

```
struct StudentType xjtuStudent[30];
```

这样，每一个数组元素都是一个结构体。

结构体的成员也可以是结构体，称为嵌套结构。例如，在下面的工资单例子中就用到了这种结构。

```
//定义日期类型
struct Date
{
    int da_year;
    int da_mon;
    int da_day;
```

```
};
//定义家庭情况类型
struct Family_type
{
    intin_double_harness;                        //婚姻状况
    int is_colleague;                            //是否双职工
    Date birthdate_of_singleton;                 //独生子女出生日期
    int children_in_school;                      //上学子女数
    int housing_area;                            //住房面积
};
//定义工资单类型
struct Salary_type
{
    char                 department[11];         //工作部门
    char                 name[9];                //姓名
    int                  position;               //职务
    struct Date          date_of_work;           //参加工作时间
    struct Family_type   family;                 //家庭情况
    float                salary;                 //基本工资
    float                subsidy;                //岗位津贴
    float                insurance;              //劳保福利
    float                child_allowance;        //独生子女费
    float                rent;                   //房租
    float                cost_of_elec;           //电费
    float                cost_of_water;          //水费
    float                cost_of_heating;        //取暖费
    float                cost_of_education;      //保育费
    float                realsum;                //实发工资
};
```

通过使用日期和家庭情况类型的嵌套,工资单的总体结构更加清晰。

嵌套结构的成员访问是很简单的,对其中每个结构成员都是从外向内引用的,例如:

```
struct Salary_type ctecSalary [50];              //声明了 50 个工资单记录
```

则语句

```
ctecSalary [17].date_of_work.da_year=1997;
```

就将第 18 名员工的参加工作年份设置为 1997 年。

6.4　数组应用示例

【例 6-5】 编写一个程序,实现矩阵相乘运算。

设有 L 行 M 列矩阵 $\boldsymbol{A}_{L\times M}$和 M 行 N 列矩阵 $\boldsymbol{B}_{M\times N}$(即第一矩阵的列数等于第二矩阵

的行数)，则由线性代数得知，其积为一个 L 行 N 列的矩阵 $\boldsymbol{C}_{L\times N}$：

$$\boldsymbol{C}_{L\times N} = \boldsymbol{A}_{L\times M} \times \boldsymbol{B}_{M\times N}$$

其中，

$$C_{ij} = \sum_{k=1}^{M} A_{ik} \times B_{kj}, \quad i = 1,2,\cdots,L;\ j = 1,2,\cdots,N$$

算法：用两重循环实现对 C_{ij} 的求值。

```
for(i=0;i<L;i=i+1)
    for(j=0;i<N;j=j+1)
        求 Cij;
```

其中“求 C_{ij}”又可以细化为

```
Cij=0;
for(k=0;k<M;k=k+1)
Cij=Cij+Aik×Bkj
```

程序：

```
#include <stdio.h>
#define L 4
#define M 5
#define N 3
int main()
{
    double a[L*M]=
    {
        1.0,3.0,-2.0,0.0,4.0,
        -2.0,-1.0,5.0,-7.0,2.0,
        0.0,8.0,4.0,1.0,-5.0,
        3.0,-3.0,2.0,-4.0,1.0
    };
    double b[M*N]=
    {
        4.0,5.0,-1.0,
        2.0,-2.0,6.0,
        7.0,8.0,1.0,
        0.0,3.0,-5.0,
        9.0,8.0,-6.0
    };
    double c[L*N];
    int i,j,k;
    for(i=0;i<L;i=i+1)
        for(j=0;j<N;j=j+1)
        {
```

```
            c[i * N+ j]= 0;
                for (k= 0;k< M;k= k+ 1)
                    c[i * N+ j]= c[i * N+ j] + a[i * M+ k] * b[k * N+ j];
        }
    printf("The result is c= \n");
    for(i= 0;i< L;i= i+ 1)
    {
        for(int j= 0;j< N;j= j+ 1)
            printf("%lf ",c[i * N+ j]);
        printf("\n");
    }
    return 0;
}
```

输出：

```
The result is c=
 32    15   - 9
 43    27    24
- 1   - 21   77
 29    33   - 5
```

分析：由于C语言要求在定义二维数组时要明确写出第二维的长度，这不利于将来编写通用的计算函数，所以在程序中用一维数组模拟二维矩阵，计算了4行5列矩阵 ***a*** 和5行3列矩阵 ***b*** 的乘积。这两个矩阵的数据用赋初值的方法提供。对于矩阵（二维数组）来说，输出应使用两重循环实现。

【例 6-6】 编写一个字符串处理程序，将一个字符串中的所有小写字母转换为相应的大写字母。

算法：在ASCII表中所有的大写字母从A到Z是连续排列的，所有的小写字母从a到z也是连续排列的，但大写字母和小写字母并没有排在一起。因此，如果一个字符是小写字母，就可以通过对其ASCII码作如下运算将其转换为对应的大写字母的ASCII码：

小写字母的ASCII码值－'a'＋'A'＝对应的大写字母的ASCII码值

程序：

```
#include <stdio.h>
int main()
{
    char str[]= "This is a sample";
    printf("The original string is: %s\n",str);
    int i= 0;
    while(str[i]!= 0)
    {
        if(str[i]>= 'a' && str[i]<= 'z')
            str[i]= str[i]- 'a'+ 'A';
```

```
            i=i+1;
        }
        printf("After transform: %s\n");
        return 0;
    }
```

输出：

```
The original string is: This is a sample
After transform: THIS IS A SAMPLE
```

分析：字符数据以整型或字符型格式存放，实际存放的是字符的 ASCII 码，所以可以使用数值数据的运算方法来处理字符数据。在程序中，如果一个字符不是小写字母，则不进行转换。程序中还使用了字符类型的数组，用来存放字符串，数组中的每个元素用于存放一个字符。

【例 6-7】 使用数组编写一个统计学生课程平均分的程序：

输入 6 个学生的学号和 3 门课程的成绩(整型)，统计每个学生 3 门课程的平均分，最后输出统计结果。输出格式：

```
学号  高数  英语  体育  平均分
------------------------------
```

算法：定义二维数组 student[6][5]，其中，给数组 student 前 4 列元素读值，第 1 列为学号，第 2 列到第 4 列为 4 门课程的成绩。第 5 列为平均分，通过计算求得。

程序：

```
#include <stdio.h>
#define PERSON 6
#define COURSE 3
int main()
{
    int student[PERSON][COURSE+2];
    int i,j;
    printf("Please input data of student :\n");
    for(i=0;i<PERSON;i=i+1)
    {
        scanf("%d",&student[i][0]);
        student[i][COURSE+1]=0;
        for(j=1;j<=COURSE;j=j+1)
        {
            scanf("%d",&student[i][j]);
            student[i][COURSE+1]=student[i][COURSE+1]+student[i][j];
        }
        student[i][COURSE+1]=student[i][COURSE+1]/COURSE;
    }
```

```
    printf("学号  高数  英语  体育  平均分\n");
    printf("-----------------------------\n");
    for(i=0;i<PERSON;i=i+1)
    {
        for(j=0;j<=COURSE+1;j=j+1)
            printf("%d\t",student[i][j]);
        printf("\n");
    }
    return 0;
}
```

输入：

```
2004001 80 90 100
2004002 60 80 70
2004003 85 92 87
2004004 72 75 80
2004005 95 96 92
2004006 20 25 100
```

输出：

```
学号      高数    英语    体育    平均分
---------------------------------------
2004001   80      90      100     90
2004002   60      80      70      70
2004002   85      92      87      88
2004004   72      75      80      75
2004005   95      96      92      94
2004006   20      25      100     48
```

【例 6-8】 使用结构体重新编写例 6-7 的程序。

算法：定义一个结构体类型 StudentType，其中包含学号、各门课程成绩和平均分等数据成员，其值分别通过输入和计算求得。

程序：

```
#include <stdio.h>
#define PERSON 6
#define COURSE 3
struct StudentType
{
    char id[10];                    //学号
    int score[COURSE];              //课程成绩
    int GPA;                        //平均分
};
int main()
```

```
{
    struct StudentType xjtuStudent[PERSON];
    int i,j;
    printf("Please input data of student :\n");
    for(i=0;i<PERSON;i=i+1)
    {
        scanf("%s",&xjtuStudent[i].id);
        xjtuStudent[i].GPA=0;
        for(j=0;j<COURSE;j=j+1)
        {
            scanf("%d",&xjtuStudent[i].score);
            xjtuStudent[i].GPA=xjtuStudent[i].GPA+xjtuStudent[i].score[j];
        }
        xjtuStudent[i].GPA=xjtuStudent[i].GPA / COURSE;
    }
    printf("学号   高数   英语   体育   平均分\n");
    printf("-----------------------------\n");
    for(i=0;i<PERSON;i=i+1)
    {
        printf("%d\t",xjtuStudent[i].id);
        for(j=0;j<COURSE;j++)
            printf("%d\t",xjtuStudent[i].score[j]);
        printf("%d\n",xjtuStudent[i].GPA);
    }
    return 0;
}
```

输入:(同例 6-6)

输出:(同例 6-6)

【例 6-9】 Josephus 问题。一群小孩围坐成一圈,现在任意取一个数 n,从当前编号为 1 的孩子开始数起,依次数到 n(因为围成了一圈,所以可以不停地数下去),这时被数到 n 的孩子离开,然后圈子缩小一点。如此重复进行,小孩数不断减少,圈子也不断缩小。最后所剩的那个小孩就是胜利者。请找出这个胜利者。

算法:先定义一个表示小孩的数组,数组的值表示小孩的编号,一旦为 0 即表示被剔除。为表示小孩围成圈,可以用求模运算来使数组的遍历从尾部回到头部,从而继续计数过程。

程序:

```
#include <stdio.h>
#define Total 7                               //小孩总数
int main()
{
    int ChooseNum;                            //用户随机选取的数
```

```
    int boy[Total];                                 //表示小孩的数组
    for(int i=0;i<Total;i++) boy[i]=i+1;            //给小孩编号
    printf("Please input the number which choose to eliminate: \n");
    scanf("%d",&ChooseNum);                         //用户随机输入一个剔除的数
    printf("The boys before eliminated are:\n");
    for(i=0;i<Total;i++)
        printf("%d\t",boy[i]);
    printf("\n");
    int k=1;                          //第 k个离开的小孩
    int n=-1;                              //数组下标,下一个为 0表示从第一个孩子开始数数
    while(true)
    {
        //在圈中开始剔除
        for(int j=0;j<ChooseNum;)
        {
            n=(n+1)%Total;
            if(boy[n]!=0)j++;              //如果该小孩还在圈中,则参加计数
        }
        if(k==Total)break;                 //如果已经全部剔除完成,则跳出循环
        boy[n]=0;
        printf("After %d times eliminated,the boys left are:\n",k);
        for(i=0;i<Total;i++)
            if(boy[i]!=0) printf("%d\t",boy[i]);
        printf("\n");
        k++;
    }
    //break语句跳转至此,输出胜利者编号
    printf("The No.%d boy is the winner.\n",boy[n]);
    return 0;
}
```

6.5 函　　数

C 程序是一个或多个函数的集合。即使是最简单的程序,也会有一个 main 函数。因此,无论某个 C 程序多么复杂,规模有多么大,程序的设计最终都要落实到一个个函数的设计和编写上。

在 C 语言中,函数是构成程序的基本模块,每个函数具有相对独立的功能。函数有 3 种:主函数(即 main 函数)、C 语言提供的已经作为系统一部分的库函数和用户自己定义的函数。

合理地编写用户自定义函数,可以简化程序模块的结构,便于阅读和调试,是结构化程序设计方法的主要内容之一。

6.5.1 函数的定义

函数必须先定义才能使用。

所谓定义函数，就是编写完成函数功能的程序块。定义函数的一般格式为

```
<函数值类型标识符>函数名(<形式参数表>)
{
    <函数体>
}
```

其中：

(1) 函数名：要定义的函数的名字，它的命名应符合 C 语言对标识符的规定。在函数名后面必须有一对圆括号。

(2) 函数值类型标识符：即调用该函数后所得到的函数值的类型。例如，例 6-3 中的函数 mystrlen 的函数值的类型是 int。函数值是通过函数体内部的 return 语句提供的，其格式为

```
return <表达式>;
```

return 的功能有二：一是使流程返回调用函数，宣告函数的一次执行终结，在调用期间所分配的变量单元被释放；二是把函数值送到调用表达式中。

在编写函数时要注意，用 return 语句提供的函数值的类型应与函数说明中的函数值类型一致，否则会出现错误。

有些函数可能没有函数值，或者说其函数值对调用者来说是不重要的。这时调用该函数实际上是为了得到运行该函数内部的程序段的其他效果。这一点与数学中的函数概念有所不同，需特别注意。如果要说明一个函数确实没有返回值，可以使用说明符 void。例如，主函数

```
void main()
{
    ……
}
```

既没有返回值，也不需要参数。但要注意，这时函数中不能出现有返回值的 return 语句。

(3) 形式参数表：形式参数放在函数名后面的一对圆括号内，其作用如下：

① 表示将从主调函数中接收哪些类型的数据。例如：

```
double grav(double m1,double m2,double distance)
```

将从调用函数中接收 3 个 double 类型的数据，分别赋给变量 m1、m2 和 distance。

② 形式参数可以在函数体中引用，可以输入、输出、赋值或参与运算。有些函数不带形式参数，因此函数名后面的括号为空，但一对圆括号不能省略。

C 语言函数的参数声明格式为

<类型><参数 1>,<类型><参数 2>,…,<类型><参数 n>

例如：

```
int array[],int count
```

圆括号中的形式参数是函数与外界联系的接口，必须明确指出形式参数的名字和类型。

(4) 函数体：由一对花括号括起来的语句序列(包括变量声明)，这些语句实现函数的功能。实际上，函数体是一个分程序结构，由语句和其他分程序组成。在函数体中定义的变量只有在执行函数时才存在。

C语言的语句可以分为声明语句和执行语句两类，在一个函数体这两种语句可以交替出现，但对某具体变量来说，应先声明，后使用。

【例 6-10】 编写一个求阶乘 $n!$ 的函数。

算法：阶乘 $n!$ 的定义为

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

且规定 $0! = 1$。

程序：

```
int fac(int n)
{
    int result=1;
    if(n<0)
        return -1;
    else if(n==0)
        return 1;
    while (n>1)
    {
        result *=n;
        n--;
    }
    return result;
}
```

分析：如果 n 为负数，则函数 fac 返回 -1，负值在正常的阶乘值中是不会出现的，正好用作参数错误的标志。

该函数定义了阶乘的算法。该函数一经定义，就可以在程序中多次地使用它。函数的使用是通过函数调用来实现的。

6.5.2 函数的调用

在C程序中，除了 main 函数以外，任何一个函数都不能独立地在程序中存在。任一函数的执行都是通过在 main 函数中直接或间接地调用该函数来开始的。

函数调用的一般形式为

<函数名>(<实参表>)

函数的调用既可以出现在表达式可出现的任何地方,也可以以函数调用语句(后加分号)的形式独立出现。实参表是调用函数时所提供的实际参数值,这些参数值可以是常量、变量或者表达式。调用函数时提供给函数的实参应该与函数的形式参数表中的参数的个数和类型一一对应,这称为"虚实结合",这时形式参数从实参得到值。

【例 6-11】 阶乘函数的调用。

程序:

```
#include <stdio.h>
int main()
{
    int n;
    printf("Please input a number n to calculte n!:\n");
    scanf("%d",&n);
    printf("%d!=%d\n",n,fac(n));
    return 0;
}
```

输入:

```
Please input a number n to calculte n! : 5
```

输出:

```
5!=120
```

一个 C 程序经过编译以后生成可执行的代码,形成后缀为 exe 的文件,存放在外存储器中。当程序被启动时,首先从外存将程序代码载入到内存的代码区,然后从入口地址(main 函数的起始处)开始执行。程序在执行过程中,如果遇到了对其他函数的调用,则暂停当前函数的执行,保存下一条指令的地址(即返回地址,作为从子函数返回后继续执行的入口点),并保存现场,然后转到子函数的入口地址,执行子函数。当遇到 return 语句或子函数结束时,则恢复先前保存的现场,并从先前保存的返回地址开始继续执行。图 6-6 说明了函数调用和返回的过程,图中的标号标明了执行顺序。

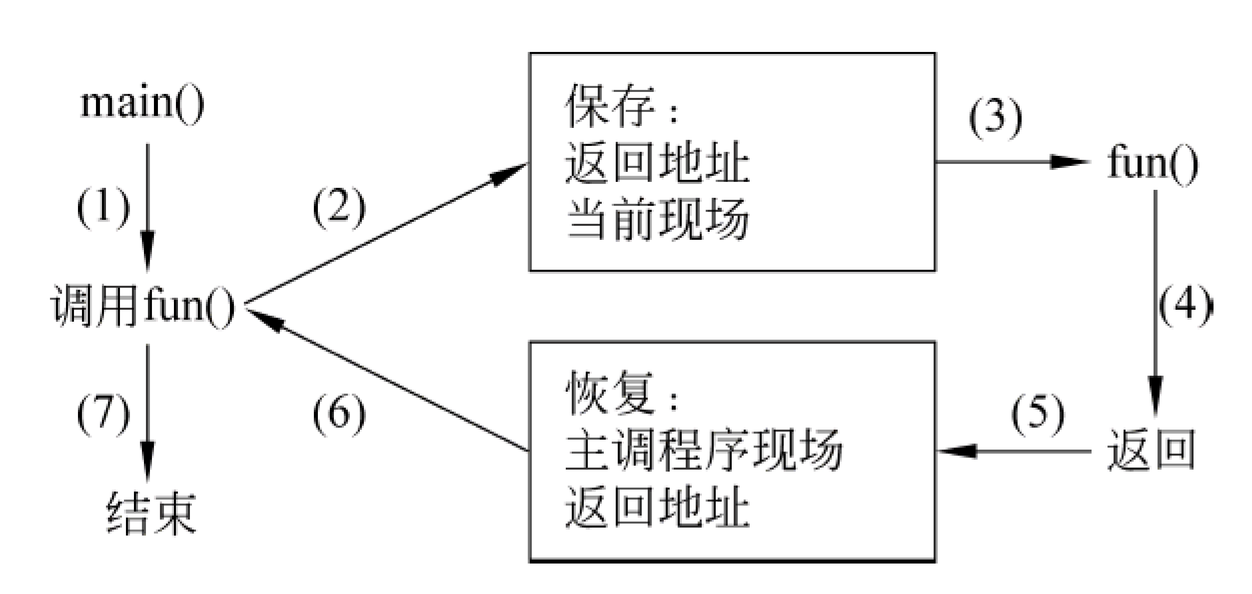


图 6-6 函数调用和返回的过程

6.5.3　函数原型

C 语言规定，函数和变量一样，在使用之前也应该事先说明。函数的定义可视为对函数的说明。因此，在前面的例子中，函数的定义均放在程序的前部。另外，在 C 语言中还有一种函数的引用性说明，即函数原型，通常也称其为函数声明。函数原型的一般格式为

<函数返回值的类型标识符>函数名(<形式参数表>);

其中各部分的意义与函数定义相同。

函数原型与函数定义的区别在于：函数原型没有函数体部分，且是用分号结束的，就像变量的说明一样。

有了函数原型，即使函数的定义放在其引用之后，只要将函数原型放在对函数的调用之前，因为函数原型向编译器提供了函数的名称、函数返回值类型和参数的个数、顺序及类型等信息，所以也就不会引起编译失败。

【例 6-12】　求两数中较大的数(函数原型的使用)。

程序：

```
#include <stdio.h>
int max(int x,int y);                  //函数原型
int main()
{
    printf("Enter two integers: \n");
    int a,b;
    scanf("%d"%d",&a,&b);
    printf(" The larger number is %d\n",max(a,b));
    return 0;
}
//函数定义
int max(int x,int y)
{
    return x>y?x:y;
}
```

分析：尽管函数 max 的定义出现在对它的调用之后，但由于使用了函数原型，程序就能成功地编译通过。

在函数原型中，参数的名字也可以省略，也就是说可以不必指定参数列表的变量名，但必须指定每一个参数的数据类型。因此，上例中的函数原型也可以改写为

```
int max (int,int);
```

6.5.4　函数间的参数传递

由于函数通常是用于实现一个具体功能的模块，所以它必然要和程序中的其他模块

交换信息。实际上,一个函数可以从函数之外获得一些数据,并可向其调用者返回一些数据。这些数据主要是通过函数的参数与函数的返回值来传递的。

在 C 语言中,实参与形参有两种结合方式:值调用和地址调用。本节介绍值调用,6.9 节介绍地址调用。

值调用的特点是调用时实参仅将其值赋给了形参,因此,在函数中对形参值的任何修改都不会影响到实参的值。前面介绍的例子中的函数调用均为值调用。值调用的好处是减少了调用函数与被调用函数之间的数据依赖,增强了函数自身的独立性。

【例 6-13】 交换两个变量的值。

程序:

```
//交换两个变量的值 (不成功)
#include <stdio.h>
void swap(int x,int y)
{
    int tmp;
    tmp=x;
    x=y;
    y=tmp;
}
//测试函数 swap用的主函数
int main()
{
    int a=1,b=2;
    printf("Before exchange: a=%d",b=%d"\n",a,b);
    swap(a,b);
    printf("After exchange: a=%d",b=%d"\n",a,b);
    return 0;
}
```

输出:

```
Before exchange: a=1,b=2
After exchange: a=1,b=2
```

分析:从输出结果来看,函数 swap 并没有完成交换两个变量的任务。为什么?如前所述,函数的参数实际上相当于在函数内部声明的变量,只是在调用时由实参变量 a 和 b 为其提供初值。因此,虽然在函数 swap 中变量 x 和 y 的值确实被交换了,但它们对在主函数中作为调用函数 swap 的实参的 a 和 b 却并无影响。考虑用如下语句调用 swap 函数的情况:

```
swap(2,3+a);
```

这一点就更加明显了:常数 2 和表达式 $3+a$ 用于向 swap 函数的参数 x 和 y 传递初值,而常数 2 和表达式 $3+a$ 交换是没有意义的。

6.5.5 局部变量和全局变量

作用域是指程序中使一个标识符的有意义的一段区域，该标识符在该段区域是可见的，或者说是在该区域内是可以使用该标识符的。

根据作用域的不同，可以将 C 程序中的变量分为局部变量和全局变量。局部变量是在函数或分程序中声明的变量，只能在本函数或分程序的范围内使用。而全局变量声明于所有函数之外，可以为本源程序文件中位于该全局变量声明之后的所有函数共同使用。

全局变量的用途是在各个函数之间建立某种数据传输通道。通常，使用返回值和参数表在函数之间传递数据，这样做的好处是数据流向清晰自然，易于控制，数据也较为安全。但有时会遇到这种情况：某个数据为许多函数所共用，为了简化函数的参数表，可以将其说明为全局变量。

初看起来，全局变量可以为所有的函数所共用，使用灵活方便，因此颇为一些初学者所喜爱，在程序中大量使用。实际上，滥用全局变量会破坏程序的模块化结构，使程序难于理解和调试。因此要尽量少用或不用全局变量。

如果在一段程序中，既有全局变量，也有局部变量，而且全局变量和局部变量的变量名相同，这时会出现什么情况呢？请看下面的例子。

```
#include <stdio.h>
int x;                              //声明全局变量
int func1(int x)                    //函数 func1 有一个名为 x 的参数
{
    return (x+5) * (x+5);
}
int func2(int y)
{
    int x=y+5;                      //函数 func2 中声明了一个名为 x 的局部变量
    return x * x;
}
int main()
{
    x=0;                            //在主函数中为全局变量 x 赋值
    printf("The result in func1 :%d\n",func1(5));
    printf("The result in func2 :%d\n",func1(2));
    printf("x=%d\n",x);
    return 0;
}
```

在上面的程序中一共有 3 个变量 x：一个是全局变量，一个是函数 func1 的参数，还有一个是函数 func2 中的局部变量。虽然全局变量的作用范围是整个源程序，但就上面这段程序而言，只有在主函数中才能使用全局变量 x，而在其他两个函数中的 x 均是它们的参数或局部变量。这种现象可以用“地方保护主义”形象地说明。

6.5.6 递归函数

任何一个可以用计算机求解的问题的难度都和其问题规模有关。问题规模越大，直接解决起来就越困难，而对于规模比较小的问题，一般计算时间也较短，问题也比较容易求解。所以解决大的问题时就可以采用分治法，即将一个难以解决的大问题反复分解成规模较小的若干子问题，这些子问题与原问题基本一致，但规模却不断缩小，最终使子问题缩小到可以很容易求解的程度，然后通过解决这些子问题而求解出原来的大问题。这个过程就引出了递归算法。

当定义一个函数时，如果其函数体内有调用其自身的语句，则该函数称为递归函数。一个直接或间接地调用了自身的算法就是递归算法。在数学中，有很多问题可以用递归的方法定义。对于这类问题，用递归函数编写程序方便简洁，可读性好。编写递归函数时，只要知道递归定义的公式，再加上递归终止的条件就能容易地编写出相应的递归函数。

【例 6-14】 采用递归算法求 n!。

算法：由阶乘的概念可以写出其递归定义：

$$0! = 1$$
$$n! = n * (n-1)!$$

程序：

```
//函数 fac: 求阶乘的递归函数
int fac(int n)
{
    if(n< 0)                    //不能求负数的阶乘
        return - 1;
    else if(n==0)               //0的阶乘为 1
        return 1;
    else
        return n* fac(n-1);     //n!为 (n-1)!乘以 n
}
```

分析：用递归函数 fac 计算 5!时的执行过程如图 6-7 所示。

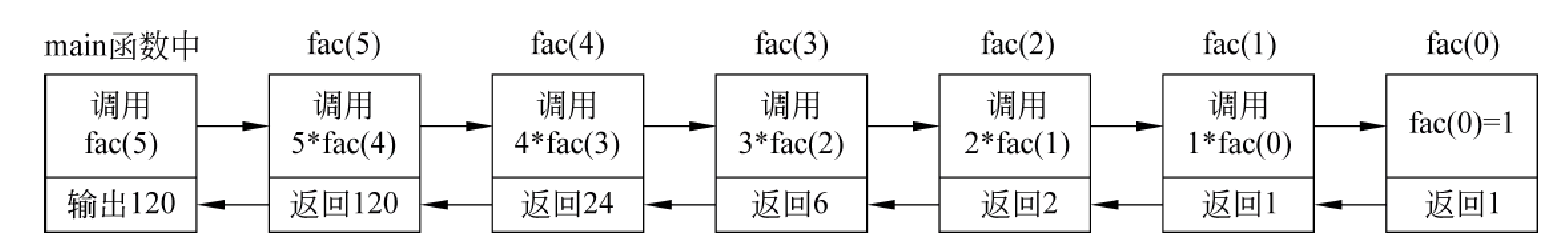


图 6-7 递归函数的调用顺序

由这个例子可以看出，一个问题要转化为递归来处理，必须满足以下条件：

（1）必须包含一种或多种非递归的基本形式。

（2）一般形式必须能最终转换到基本形式。

(3) 由基本形式来结束递归。

除了上面的阶乘问题以及习题第 26 题中的 Ackermann 函数、Fibonacci 数列等有明显递归定义的数学问题可以用递归来处理以外，还有一些问题（如八皇后问题、梵塔问题等），虽然问题本身没有明显的递归结构，但如果用递归求解将会比较简单。其中最为经典的递归问题莫过于梵塔（汉诺塔）问题。

【例 6-15】 梵塔问题。

根据古印度神话，在贝拿勒斯的圣庙里安放着一个铜板，板上插有 3 根一尺长的宝石针。印度教的主神梵天在创造世界的时候，在其中的一根针上摆了由小到大共 64 片中间有孔的金片。无论白天和黑夜，都有一位僧侣负责移动这些金片，规则是一次只能将一片金片移到另一根针上，并且在任何时候以及任一根针上，小片永远在大片的上面。当所有的 64 片金片都由最初的那根针移到另一根针上时，这世界就将在一声霹雳中消失。

现在就要编写一段程序来模拟这个过程。

算法：用字符 A、B 和 C 表示 3 根针，则梵塔问题就如图 6-8 所示。

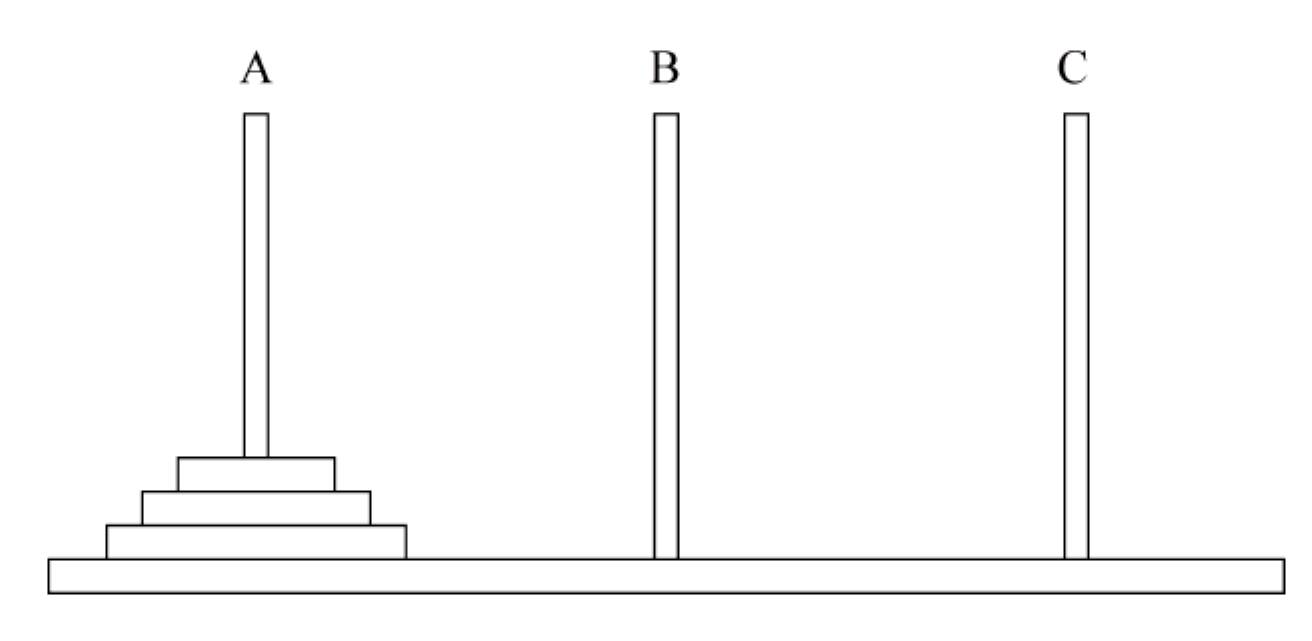


图 6-8 梵塔问题

如果只有 1 片金片时，问题就比较简单，此时，只要直接将金片从 A 针移到 C 针上即可；而当 $n>1$ 时，就需要借助另外一个针来移动。通过分析可以看出，将 n 片金片由 A 移到 C 上可以分解为以下几个步骤：

(1) 将 A 上的 $n-1$ 片金片借助 C 针移到 B 针上。

(2) 把 A 针上剩下的一片金片由 A 针移到 C 针上。

(3) 将剩下的 $n-1$ 个金片借助 A 针由 B 针移到 C 针上。

步骤(1)和(3)与整个任务类似，但涉及的金片只有 $n-1$ 个了。这是一个典型递归算法。

程序：

```
#include <stdio.h>
#define N 3                                //考察当金片数为 3个时的情况
//函数 move():将金片由一根针移到另一根针上
void move(char from,char to)
{
    printf("From %c to %c\n",from,to);
}
```

```
//函数 hanoi():将 n片金片由 p1借助 p2移到 p3上
void hanoi(int n,int p1,int p2,int p3)
{
    if(n==1)
        move(p1,p3);
    else
    {
        hanoi(n-1,p1,p3,p2);
        move(p1,p3);
        hanoi(n-1,p2,p1,p3);
    }
}
//测试用主函数
int main()
{
    hanoi(N,'A','B','C');
    return 0;
}
```

输出:

```
From A to C
From A to B
From C to B
From A to C
From B to A
From B to C
From A to C
```

分析:在程序中可以通过常量 N 确定总共要移动的金片数。在上述程序中取 N 为3,从程序的运行结果可以看出,只需7步就可以将3片金片由A针移到C针上。但是随着金片数的增加,所需步数会迅速增加。实际上,如果要将64片金片全部由A针移到C针,共需 $2^{64}-1$ 步。这个数字有多大呢?如果假定圣庙里的那位僧侣以每秒钟1次的速度移动金片,日夜不停,则需要58万亿年才能完成!即使用每秒钟运算100万次的计算机来模拟这个过程,也需5800万年(实际上所需的时间还要多些,因为模拟移动一片金片就需要计算机执行多条指令)。

许多问题既可以用递归的方法求解,也可以用循环结构求解。上述求阶乘的算法就是如此。一般来说,递归程序结构清晰、简单,容易阅读和理解。但事物往往具有两重性,递归也有其缺点。因为递归程序在执行过程中需要保存大量的中间状态,在目标代码中要通过堆栈实现,这难以事先估计需要的存储量,执行速度也比较慢,所以从运行时间及空间来看,递归算法的效率比较低。在有些情况下(如程序频繁使用的部分),为提高程序效率,必须想办法消除递归,通常的做法是采用一个用户定义的栈来模拟系统的递归调用工作栈,从而达到将递归算法转换为非递归算法的目的。感兴趣的读者请自行

参考有关文献。但也有些递归算法很难用通常的循环结构实现，例如对于树形数据结构的处理。

6.5.7 带参数的 main 函数

本书前面出现过的所有例子中 main 函数都是不带参数的，而实际上，main 函数是可以带参数的，其函数原型为

```
int main(int argc,char * argv[])
```

其中，第一个整型参数指明在以命令行方式执行本程序时所带的参数个数(包括程序名本身，故 argc 的值至少为 1)；第二个参数为一个字符型指针数组(其中第 1 个下标变量 argv[0]指向本程序名，接下来的下标变量 argv[1]，argv[2]等分别指向命令行传递给程序的各个参数)，用来存放命令行中命令字及各个参数的字符串。

【例 6-16】 带参数的 main 函数的使用。

程序：

```
//文件名设为 abc.cpp
#include <stdio.h>
int main(int a,char * ar[])
{
    if(a!=2)
    {
        printf("Error!!!\n");
        printf("Usage: ProgramName <sb's name> \n");
        return 1;
    }
    printf("Hello,%s. Welcome to the world of C++!\n",ar[1]);
    return 0;
}
```

假设本例生成的执行程序文件名为 abc.exe，在命令行输入

```
abc SNCui
```

来运行这个程序，程序的输出为

```
Hello,SNCui. Welcome to the world of C++!
```

分析：由程序执行可以看到，系统不仅在程序开始时自动调用了 main 函数，同时也自动对参数进行了赋值。

值得注意的是，在大多数系统中，如果出现多个参数，则每个命令行参数之间应该以空格或制表符分隔，而不能使用逗号、分号等其他符号。

从理论上讲，出现在命令行中的参数数目可以多达 32 767 个。

6.5.8　C语言的库函数

为了方便程序员编程，C语言提供了大量已预先编制的函数，即库函数。对于库函数，用户不用定义也不用声明就可直接使用。由于C语言软件包将不同功能的库函数的函数原型分别写在不同的头文件中，所以，用户在使用某一库函数前，必须用include预处理指令给出该函数的原型所在头文件的文件名。例如，欲使用库函数sqrt，由于该函数的原型在头文件math.h中，所以必须在程序中调用该函数前写一行

```
#include <math.h>
```

前面已经介绍了字符串处理类库函数。C语言的库函数很多，很难一一列举。学习库函数的用法，最好的方法是通过联机帮助查看该函数的声明，如果有疑问，则再编一个小验证程序实际测试其参数和返回值。

6.6　变量的存储类别

在C语言中，根据变量存在时间的不同，可以将存储类别分为4种，即自动(auto)、静态(static)、寄存器(register)和外部(extern)。

6.6.1　自动变量

自动变量和静态变量的区别在于其生存期不同。

自动变量的特点是在程序运行到自动变量的作用域(即声明了自动变量的那个函数或分程序)中时才为自动变量分配相应的存储空间，此后才能向变量中存储数据或读取变量中的数据。一旦退出声明了自动变量的那个函数或分程序，程序会立即将自动变量占用的存储空间释放，被释放的空间还可以重新分配给其他函数中声明的自动变量使用。因此自动变量的生存期从程序进入声明了该自动变量的函数或分程序开始，到程序退出该函数或分程序时结束。在此期间之外自动变量是不存在的。自动变量的初值在每次为自动变量分配存储后都要重新设置。

自动变量对存储空间的利用是动态的，通过分配和回收，不同函数中定义的自动变量可以在不同的时间中共享同一块存储空间，从而提高了存储器的利用率。显然，前面介绍的局部变量(也包括函数的参数)都是自动变量。同样显然的是在整个程序运行过程中，一个自动变量可能经历若干个生存期。而在自动变量的各个不同生存期中，程序为该变量分配的存储空间的具体地址可能并不相同，因此在编写程序时，不能期望在两次调用同一函数时，其中定义的同一个局部变量的值之间会有什么联系。

一般在函数体或程序块中声明的变量，其存储类别默认为自动变量。关键字auto也可以被用来显式地声明auto存储类别，例如：

```
auto int    x,y,z ;
auto double  a=98.0;
```

6.6.2 静态变量

静态变量的特点是在程序开始运行之前就为其分配相应的存储空间,在程序的整个运行期间,静态变量一直占用着这些存储空间,直到整个程序运行结束。因此静态变量的生存期就是整个程序的运行期。另外,在主函数的开始说明的局部变量也具有和整个程序运行期相同的生存期。如果在声明静态变量的同时还声明了初值,则该初值也是在分配存储空间的同时设置的,以后在程序的运行期间不再重复设置。

如果需要在函数中保留一些变量的值,以便下次进入该函数以后仍然可以继续使用。但又不想使用全局变量(因为全局变量会使程序变得难于阅读、难于调试)。此时可以将该变量说明为静态局部变量。其说明格式是在原来的变量说明语句前面加上 static 构成,如下面的例子。

【例 6-17】 静态局部变量的使用。

程序:

```
#include <stdio.h>
using namespace std;
int func()
{
    static int count=0;
    return ++count;
}
int main()
{
    for(int i=0;i<10;i++)
        printf("%d\t",func());
    printf("\n");
    return 0;
}
```

输出:

```
1    2    3    4    5    6    7    8    9    10
```

分析:函数 func 中声明了一个静态局部变量 count。静态局部变量同时具有局部变量和自动变量的特点。静态局部变量 count 只能在其定义域(函数 func)中使用,但其生存期却与整个程序的运行期相同。程序在运行之前就为该变量分配了相应的存储空间并赋了初值 0,以后每次进入函数 func 时可以对其进行操作,但在离开函数 func 后 count 占用的存储空间并不释放,其中的内容也就不会发生变化,直到下一次进入该函数后又可以继续使用该变量了。静态变量的初值是在程序开始运行之前一次设置好的,不像自动

变量，每次为其分配存储空间时都要重新设置一次。

函数 func 中的局部变量 count 的作用是统计调用函数 func 的次数。

6.6.3 寄存器变量

所谓寄存器变量，即为该变量分配的存储空间并不在内存储器中，而是 CPU 中的某个寄存器。如果某寄存器分配给一个变量，则由于该变量中的数据无须再去内存中存取，所以速度很快。但由于通常计算机中寄存器的数目很少（例如常见的 PC 系列微型计算机中共有 16 个通用寄存器），使用又很频繁，所以只有那些使用最多的变量才应该说明为寄存器变量。尽管如此，C 语言规定，程序中只能定义整型寄存器变量（包括 char 型、int 型和指针变量），而且程序员定义的寄存器变量并不一定都要分配寄存器，C 语言的编译程序有权在寄存器不敷分配时将一些或全部寄存器变量自动转换为一般的自动变量或静态变量。寄存器变量的说明方法为在原来的变量说明语句之前加上 register，例如：

```
register int i,j;
```

注意：只有局部自动变量和形参可以作为寄存器变量，其他变量如全局变量、局部静态变量都不能作为寄存器变量出现。

6.6.4 外部变量

使用 extern 可以说明某一个变量为已定义外部变量，这时处于某一函数中的该变量就是外部变量，而非一个同名的局部变量。其格式为

```
extern <类型说明符><变量名表>;
```

具体应用见 6.6.5 节的例子。

6.6.5 多源程序文件程序中的全局变量说明

C 程序是由函数组成的模块化结构程序。组成一个程序的所有函数模块可以都放在同一个源程序文件中，也可分别放在几个不同的源程序文件中。在编译时，同一个源程序文件中的函数模块被编译成一个目标文件，所以一个比较大的程序可能包含若干个目标文件，在连接时再将这些目标文件组装成一个运行文件。

如果组成一个程序的几个源程序文件中都要用到同一个全局变量，应该在哪个源程序文件中说明这个全局变量呢？这时可以任选一个源程序文件，在其中说明该全局变量，而在其他的源程序文件中用外部说明语句说明该变量为外部变量。

有的时候，程序员希望某源程序文件中定义的全局变量仅限于该源程序文件中的各函数使用，即说明局限于该源文件的全局变量，这时可以在定义全局变量的类型符前加一个 static 保留字。例如，如果某源程序文件中有如下全局变量说明：

```
static int a,b,c;
```

则表明 a、b、c 三个全局变量只能在本源程序文件中使用。图 6-9 说明了在多源程序文件程序中全局变量的作用范围。

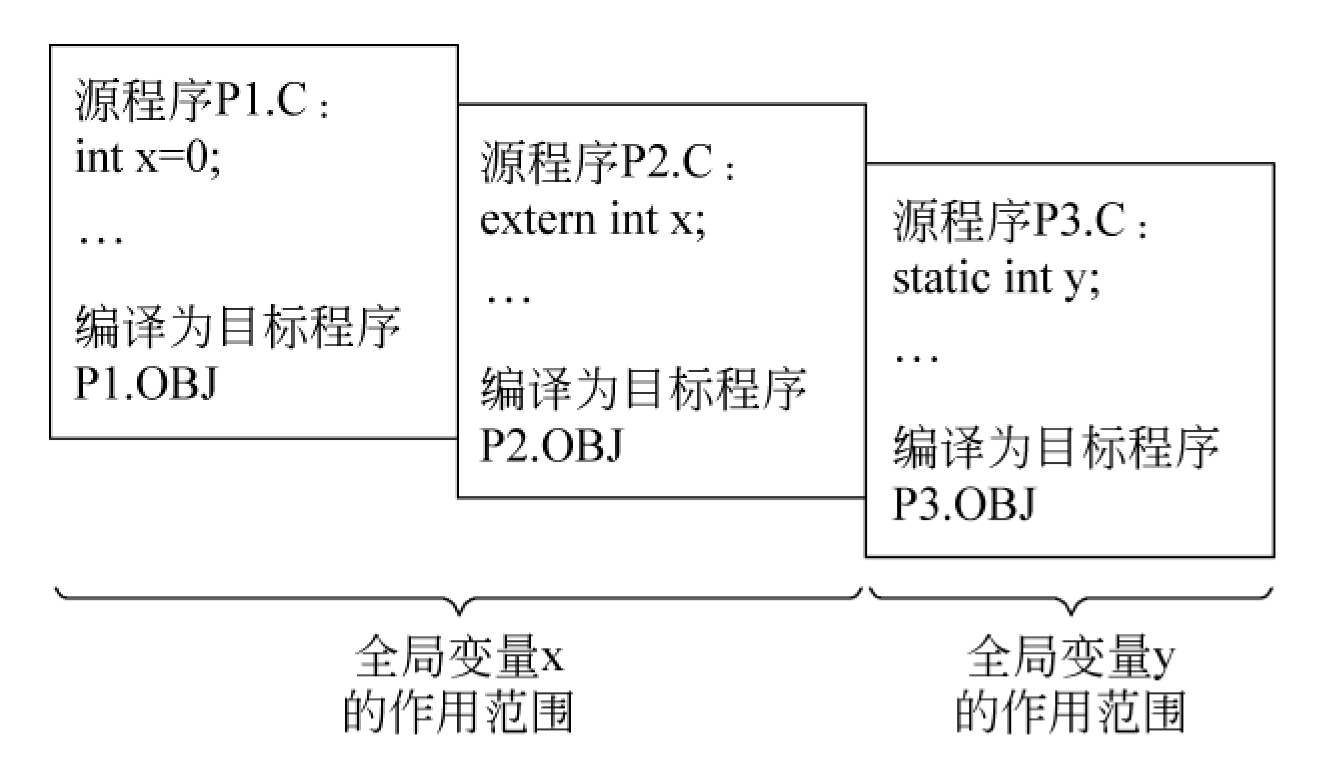


图 6-9　多源程序文件程序中全局变量的作用范围

在图 6-9 中，整个程序由 P1. C、P2. C 和 P3. C 三个源程序文件组成，编译后分别生成 P1. OBJ、P2. OBJ 和 P3. OBJ 三个目标文件。在源程序 P1. C 和 P3. C 中分别说明了两个全局变量 x 和 y。当然，x 的作用域为整个 P1. C，y 的作用域为整个 P3. C。在编译时生成的目标模块 P1. OBJ 和 P2. OBJ 中分别为它们分配了存储空间。而 P2. C 中的外部说明语句 extern int x；是说在源程序 P2. C 中也可以使用全局变量 x，但在 P2. OBJ 中并不为 x 分配存储，而是使用其他目标模块中的同名全局变量，即 P1. C 中的全局变量 x。由于在 P3. C 中说明全局变量 y 时使用了 static，所以即使其他源程序文件中再有外部说明语句 extern int y；也无法使用该全局变量。

C 语言中变量的使用方法简要总结如下：

(1) 最常用的变量形式是局部变量。一般的局部变量都是自动变量，其作用域为定义局部变量的函数或分程序，生存期为程序执行到变量定义域中的期间，即每次进入其定义域时才为局部变量分配存储空间，并设置初值(如果需要)。

(2) 可以通过在说明语句前面加上保留字 static 将局部变量说明为静态局部变量。静态局部变量的作用域仍为定义局部变量的函数或分程序，但其生存期扩大到整个程序的运行期，在程序运行之前即为变量分配存储并设置初值，该初值在以后每次调用该函数时不在重新设置。静态局部变量的主要用途是保存函数的执行信息。

(3) 定义于所有函数之外的变量称为全局变量，全局变量都是静态的，即具有和程序执行期相同的生存期。一般来说，全局变量的作用域还可以扩充到其他源程序中，只要在相应的源程序文件中加入外部函数说明语句 extern 即可。当然，也可以通过在说明语句之前加上 static 明确宣布某全局变量的作用域仅限于说明该变量的源程序文件中。

(4) 使用寄存器变量的运算速度很高，但通常可以使用的寄存器变量很少，且仅限于整型和指针。而且在寄存器不敷分配时，编译程序有权将寄存器变量转为一般局部或全局变量。

6.7　函数应用示例

【例 6-18】　打印 1000～10000 之间的回文数。所谓回文数是指其各位数字左右对称的整数，例如 12321、789987、1 等都是十进制回文数。

算法：判断一个数是否回文数，可以用除以 10 取余的方法，从最低位开始，依次取出该数的各位数字，然后用最低位充当最高位，按反序重新构造新的数，比较与原数是否相等，若相等，则原数是回文数。

程序：

```
#include <stdio.h>
int Ispalindrome(int n);
int main()
{
    for(int i=1000;i<10000;i++)
    {
        if(Ispalindrome(i))
            printf("%d\t",i);
    }
    printf("\n");
    return 0;
}
int Ispalindrome(int n)
{
    int k,m=0;
    k=n;
    while(k)
    {
        m=m*10+k%10;
        k=k/10;
    }
    return (m==n);
}
```

输出：

```
1001	1111	1221	1331	1441	1551	1661	1771	1881	1991
2002	2112	2222	2332	2442	2552	2662	2772	2882	2992
3003	3113	3223	3333	3443	3553	3663	3773	3883	3993
4004	4114	4224	4334	4444	4554	4664	4774	4884	4994
5005	5115	5225	5335	5445	5555	5665	5775	5885	5995
6006	6116	6226	6336	6446	6556	6666	6776	6886	6996
7007	7117	7227	7337	7447	7557	7667	7777	7887	7997
```

```
8008    8118    8228    8338    8448    8558    8668    8778    8888    8998
9009    9119    9229    9339    9449    9559    9669    9779    9889    9999
```

【例 6-19】 编写一个用于字符串比较的函数 mystrcmp。

算法：字符串的比较应按词典序判断。例如，由于单词 word 在词典中排在单词 work 的前面，所以单词 word 小于单词 work。实际上进行两个字符串的比较时，要按字符串中的各个字符在 ASCII 码表中的次序进行比较，这是因为在字符串中不仅可以出现字母，还可能出现其他符号。只有当两个字符串中所有对应位置上的符号分别相同时，才能认为这两个字符串相等。

程序：

```
#include <stdio.h>
int mystrcmp(char s1[],char s2[])
{
    int i=0;
    while(s1[i]==s2[i] && s1[i]!=0 && s2[i]!=0)
        i++;
    return s1[i]-s2[i];
}
```

分析：本例设计了一个循环，从两个字符串的第一个字符开始比较，直到出现不同的字符，或者有一个字符串已经结束为止。从程序中可以看出，这时如果两个字符串相等，则函数 mystrcmp 返回 0；如果字符串 s1 大于字符串 s2，则返回一个正数；如果字符串 s1 小于字符串 s2，则函数返回一个负数。

【例 6-20】 定义一个结构体矩形 Rectangle，根据给出的矩形左上角顶点坐标和右下角顶点坐标，计算该矩形的面积。

程序：

```
#include <stdio.h>
#include <math.h>
struct Rectangle
{
    int topleft_x;
    int topleft_y;
    int bottomright_x;
    int bottomright_y;
};
struct Rectangle Input(int x1,int y1,int x2,int y2)
{
    struct Rectangle tmp;
    tmp. topleft_x=x1;
    tmp. topleft_y=y1;
    tmp. bottomright_x=x2;
    tmp. bottomright_y=y2;
```

```
        return tmp;
}
double GetArea(struct Rectangle rect)
{
        return fabs((rect.bottomright_x-rect.topleft_x) *
        (rect. bottomright_y-rect.topleft_y));
}
int main()
{
        struct Rectangle rec;
        int tlx,tly,brx,bry;
        printf("Please input four integers for the two vertices of rectangle in the order: \n");
        printf("topleft_x topleft_y bottomright_x bottomright_y\n");
        scanf("%d %d %d %d",&tlx,&tly,&brx,&bry);
        rec=Input(tlx,tly,brx,bry);
        printf("Area=%lf\n",GetArea(rec));
        return 0;
}
```

输入：

```
0 100 200 0
```

输出：

```
Area=20000
```

分析：本程序展示了结构体变量作为函数的参数和返回值的情况。函数 Input 的返回值是一个结构体，函数 GetArea 的参数为一个结构体变量。

6.8 地址与指针

C 语言具有强大的地址运算和操作能力，这种操作地址的特殊类型的变量就是指针。指针也是 C 语言区别于其他程序设计语言的主要特性之一。正确灵活地使用指针，可以有效地表示和访问复杂的数据结构，可以动态分配内存，直接对内存地址操作，可以提高某些程序的执行效率。但同时它又是较难掌握的内容，使用时很容易出错。

6.8.1 地址

从拓扑结构上来说，计算机的内存储器（简称内存）就像一个巨大的一维数组，每个数组元素就是一个存储单元。就像数组中的每个元素都有一个下标一样，每个内存单元都有一个编号，称为地址，它可以用一个无符号整数来表示。计算机就是通过这种地址编号的方式来管理内存数据读写定位的，如图 6-10 所示。

内存是程序活动的基本场地。在运行一个程序时，程序本身及其所用到的数据都要放在内存中，如程序、函数、变量、常量、数组和对象等。凡是存放在内存中的程序和数据都有一个地址，可以用它们占用的那片存储单元中的第一个存储单元的地址表示。在 C 语言中，为某个变量或者函数分配存储器的工作由编译程序完成[①]。在编写程序时，通常是通过名字来使用一个变量或者调用某个函数，而变量和函数的名字与其实际存储地址之间的变换由编译程序自动完成，这样做既直观又方便。同时，C 语言也允许直接通过地址处理数据，在很多情况下这样做可以提高程序的运行效率。

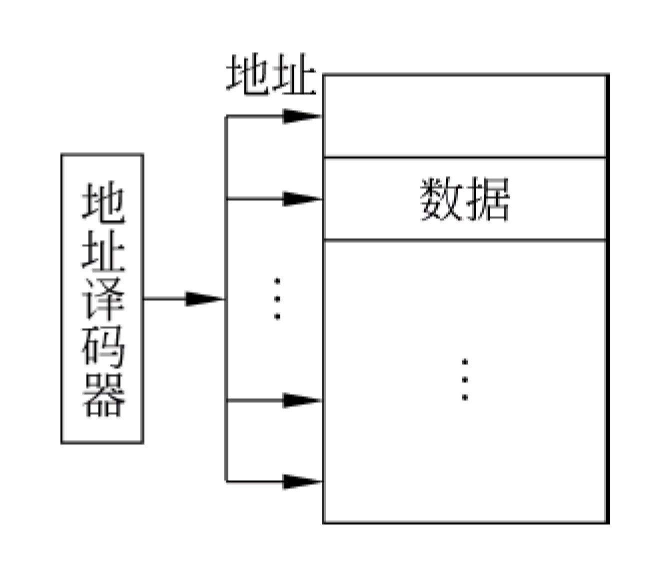


图 6-10　存储结构示意图

那么，如何知道某个变量、数组或者函数的地址呢？

C 语言规定：

(1) 变量的地址可以使用地址运算符 & 求得。例如，&x 表示变量 x 的地址。

(2) 数组的地址，即数组第一个元素的地址，可以直接用数组名表示。

(3) 函数的地址用函数名表示。

6.8.2　指针

某个变量的内存地址称为该变量的指针。因此，用以表示(或存储)不同指针值(即地址值)的变量就是指针变量，简称指针。

如上所述，指针是一个变量，因此它也具有变量的几个要素：

(1) 指针的变量名：与一般变量的命名规则相同。

(2) 指针变量的类型：不是指针自身的类型，而是指针所指向的变量的数据类型。

(3) 指针变量的值：是指针所指向的变量在内存中所处的地址。

同样，指针变量也必须遵循“先声明，后使用”的原则。指针变量的声明形式为

数据类型 * 指针变量名；

其中数据类型是指针所指向的变量的数据类型，根据指针所指向变量类型的不同，指针分为不同的类型。例如：

```
int* ptr;
```

定义了一个名为 ptr 的指针，该指针指向一个 int 型的变量。这里可以将“int *”理解为“* of int”。

实际上，整型指针是将它指向的地址所代表的字节及其后 3 个字节(共 4 个连续字节)作为一个整型数据的存储单元进行操作的，而双精度指针是将它指向的地址所代表的字节及其后 7 个字节(共 8 个连续字节)作为一个双精度数据的存储单元进行操作的。同

① 实际上，为变量和函数分配存储空间的工作是分几步完成的。在编译和连接的过程中，只给程序中的各个变量分配了相对地址。每个变量的实际存储位置要到程序运行时才能确定。如果是局部变量，其存储分配更晚，要到该局部变量所属的函数被调用时才进行。因此，同一个变量，在程序的各次运行中可能被分配在不同的存储地址上。这也是为什么通常只需要知道某变量确有一个地址，而不必关心该地址值具体是多少的原因。

理，字符型指针仅是将它指向的地址所代表的字节作为一个字符型数据的存储单元处理。

指针可以指向各种类型，包括基本类型、数组、函数、对象，甚至也可以指向指针。

6.9 指针运算

指针运算本身是比较简单的。专门的指针运算符只有 * 和 &。此外指针运算还有赋值运算、算术运算、比较运算和指针下标运算。

6.9.1 * 和 & 运算符

& 称为取地址运算符，用以返回变量的指针，即变量的地址。* 称为指针运算符，用以返回指针所指向的基类型变量的值。

注意：* 和 & 出现在声明语句中和执行语句中其含义是不同的。一元运算符 * 出现在声明语句中，在被声明的变量之前时，表示声明的是指针，例如：

```
int * ptr;                    //声明 ptr 是一个 int 型指针
```

* 出现在执行语句中或声明语句的初值表达式中，表示访问指针所指向变量的值，例如：

```
y= * ptr;                     //将指针 ptr 所指向的值赋给变量 y
```

& 出现在变量声明语句中位于被声明变量左边时，表示声明的是引用，例如：

```
int &ref;                     //声明一个 int 型的引用 ref
```

& 在给变量赋初值时，出现在赋值号右边或在执行语句中作为一元运算符出现时表示取变量的地址，例如：

```
ptr= &x;                      //取变量 x 的地址
```

假如需要将常量 2 写入整型变量 x 中，可以使用语句

```
x=2;
```

如果使用指针情况会如何呢？另设 ptr 为一个 int 型指针，则下面的语句就将常量 2 存入了变量 x 中：

```
ptr= &x;
 * ptr=2;
```

使用指针后，反倒不如原来简单，是否能说明指针毫无用处呢？当然不是，请看下一个例子。

【例 6-21】 编写用于交换两个整型变量的值的函数。

分析：在例 6-13 中曾经直接编写过交换两个整型变量的值的函数 swap，但从结果可以看出，swap 函数根本没有完成交换变量 x 和 y 的任务。为什么？请看图 6-11 中的内存分配情况。

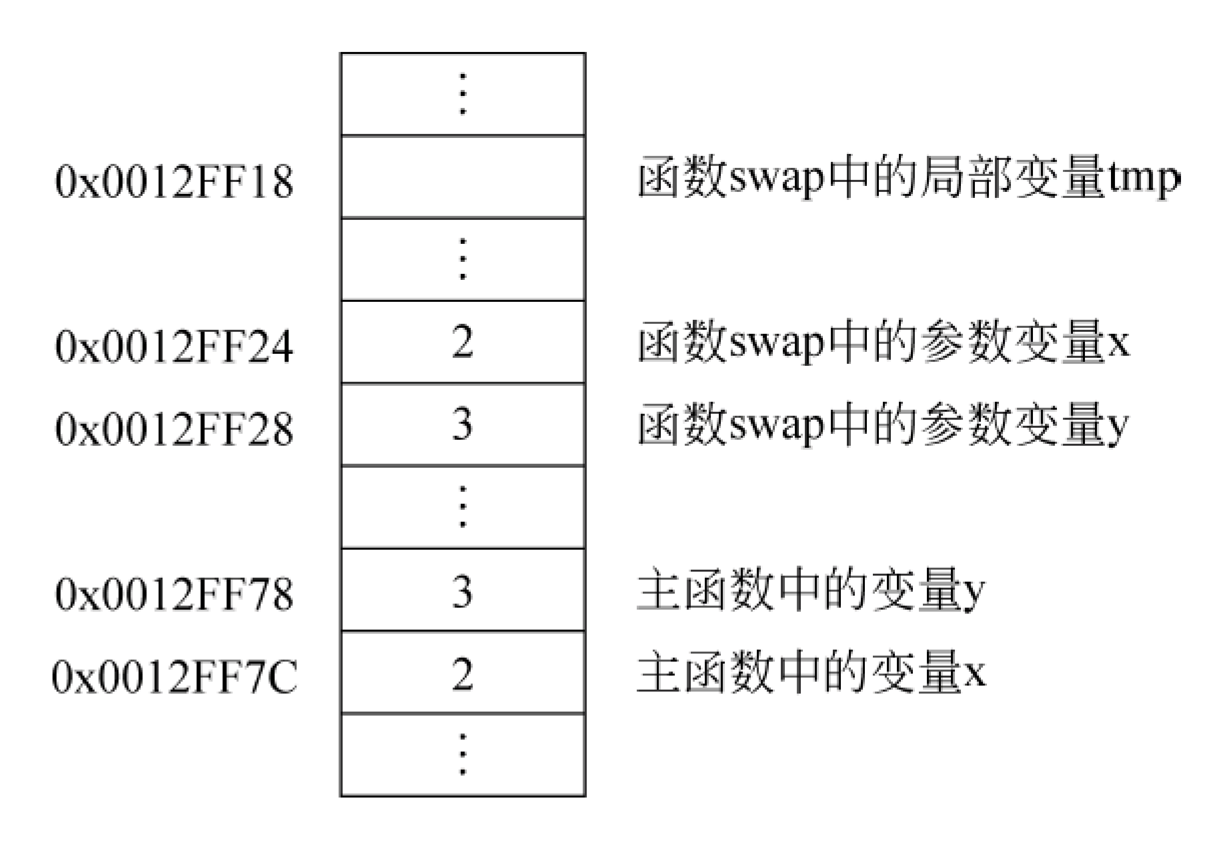


图 6-11　验证函数 swap 运算开始时的内存分配示意图[①]

图 6-11 描述的是程序运行到 swap 函数中时的内存分配。由图中可以看出，主函数中声明的变量 x 和 y 与函数 swap 的参数 x 和 y 在内存中分别占有各自的存储区，它们之间唯一的联系只是在主函数中调用函数 swap 时将主函数中变量 x 和 y 的值分别传送给函数 swap 的两个参数 x 和 y。参数 x 和 y 在函数 swap 运行期间相当于两个局部变量。因此，在函数 swap 执行完毕以后，其参变量 x 和 y 的值确实已被交换，如图 6-12 所示。

然而，事情到此就停止了。从图 6-11 中可以看出，虽然 swap 函数的两个参数变量 x 和 y 的值已被交换，但原来主函数的变量 x 和 y 的值却没有发生变化。而且，随着函数 swap 运行结束返回主函数后，swap 中为局部工作变量申请的内存空间，包括其参数 x、y 和局部变量 tmp 占用的内存单元，都将被释放，函数 swap 所做的一切工作都白费了。

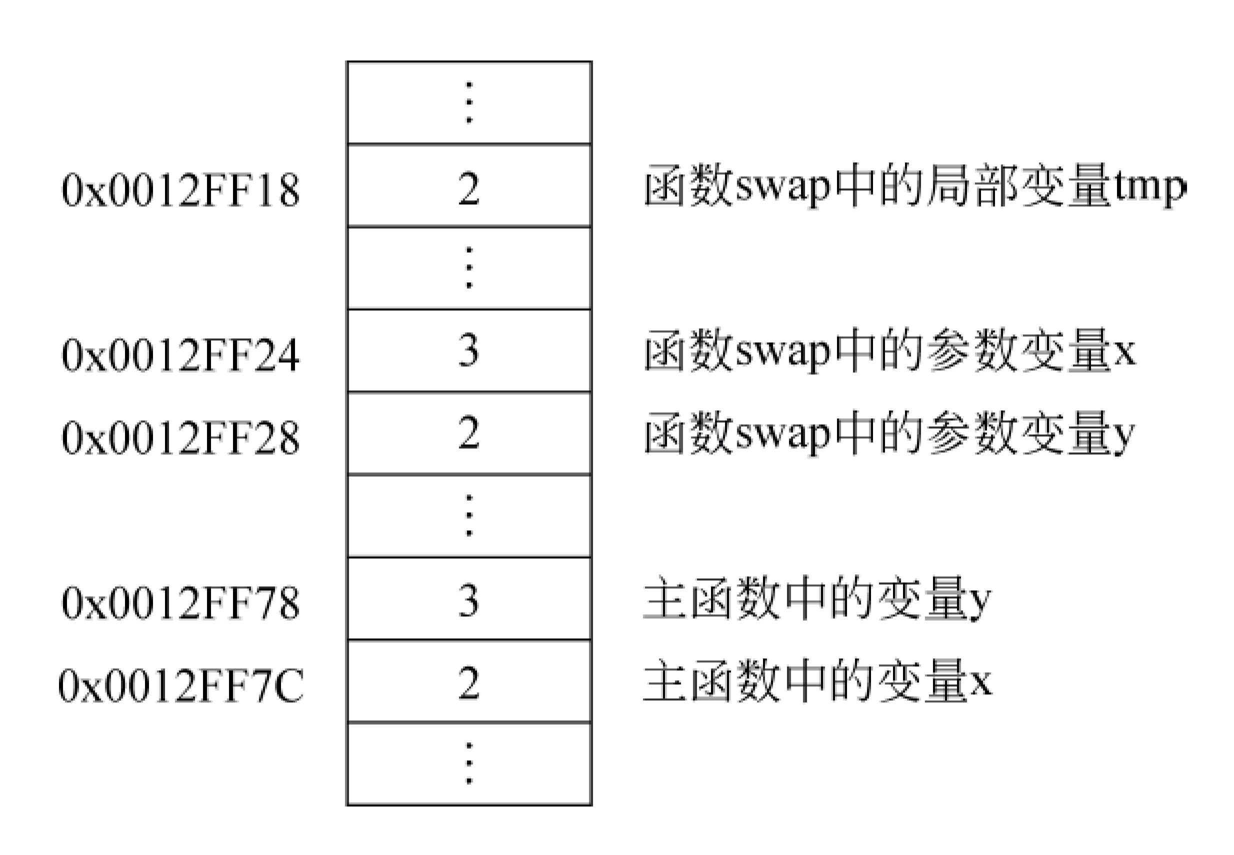


图 6-12　函数 swap 中的运算结束时的内存分配示意图

那么，怎样才能编写一个真正可以交换参数值的 swap 函数呢？这就要用到指针了。下面将函数 swap 的参数声明为指向 int 类型的指针，重新编写该函数。

---

① 由于地址分配问题相当复杂，超出本课程的范围。因此图中的地址值只是一个实例，可能会与读者调试的程序有所不同。以下各图同此。

程序：

```
//函数 swap：交换两个整型变量的值
void swap(int * xp,int * yp)
{
    int tmp;
    tmp= * xp;
     * xp= * yp;
     * yp=tmp;
}
```

注意到这次函数 swap 的参数变量 xp 和 yp 是两个指向整型变量的指针，因此在调用该函数时只能使用变量的地址作为实参。使用下列语句取代测试主函数中的函数调用语句：

```
swap(&x,&y);
```

此时的内存分配如图 6-13 所示。

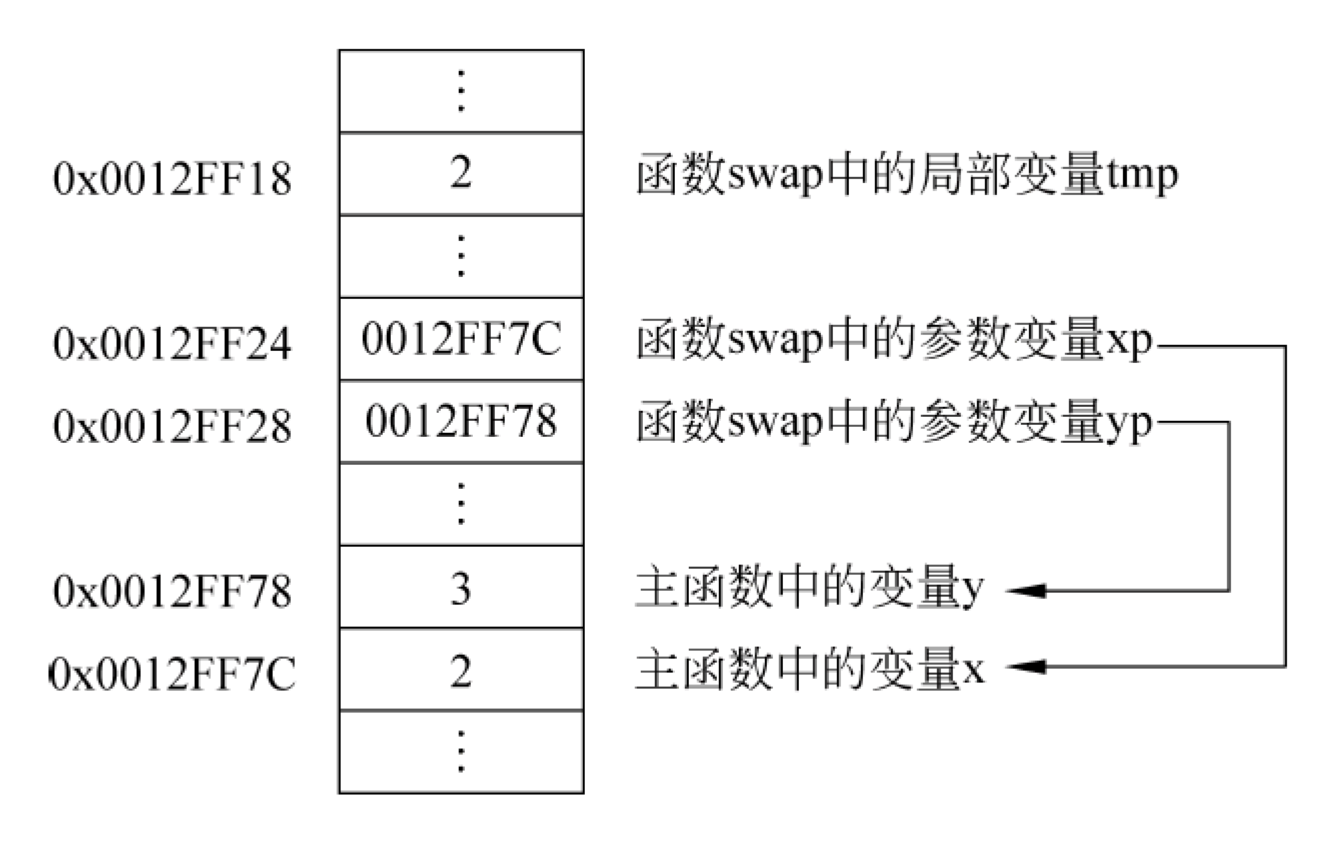


图 6-13　采用指针后函数 swap 中的运算结束时的内存分配示意图

于是，在新的 swap 函数中，以下语句

```
tmp= * xp;
 * xp= * yp;
 * yp=tmp;
```

通过地址直接对主函数中原来的变量 x 和 y 进行操作，完成了交换这两个变量的值的任务。在函数 swap 执行完毕后，即使释放其局部变量 tmp 和指针参数 xp、yp 占用的存储单元也不会影响到主函数中变量 x 和 y 的新内容。

6.9.2　指针变量算术运算

指针变量只有加法和减法两种算术运算。

（1）自增＋＋、自减－－运算。

（2）加、减整型数据。

(3) 指向同一个数组的不同数组元素的指针之间的减法。

对指针变量进行下列算术运算毫无意义：指针间相乘或相除，两个指针相加，指针与浮点型数的加、减等。

6.9.3 指针变量比较运算

在关系表达式中允许指针的比较运算，但要注意这种运算对程序设计是否有意义。一般地，指针的比较常用于两个或两个以上指针变量都指向同一个公共数据对象的情况，如同一个数组中各数组元素的指针之间的比较等。任何指针与空指针(NULL)的比较在程序设计中是必要的，但类型不同的指针之间的比较一般是没有意义的。

6.9.4 指针变量下标运算

C 语言提供了指针变量的下标运算[]，其形式类似于一维数组元素的下标访问形式。例如，在声明了指针变量

```
double x,a[100], * ptr=a;
```

之后，也可以使用

```
x=ptr[10];
```

这样的用法，并不表示 ptr 是一个数组，而只是

```
x= * (ptr+10);
```

的另一种写法。

6.10 指针与数组

6.10.1 指向数组的指针

在 C 语言中，指针与数组的关系十分密切，实际上数组名本身就是一个常量指针(所谓常量指针是说指针所指的地方保持不变)。当定义数组时，其首地址就已经确定不再改变了。例如，对于数组 array[10]，其数组名 array 就等效于地址 &array[0]。可以将 array 看作一个指针，它永远指向 array[0]。

由于数组中的元素在内存中是连续排列存放的，所以任何能由数组下标完成的操作都可由指针来实现。指向数组中元素的指针称为数组指针。使用数组指针的主要原因是能够使程序效率高，执行速度快。

设有指针 ptr、qtr 以及字符型数组 string：

```
char * ptr, * qtr;
```

```
char string[6]="Big";
int len=strlen(string);
ptr=string;
qtr=ptr+len;
```

其关系如图 6-14 所示。

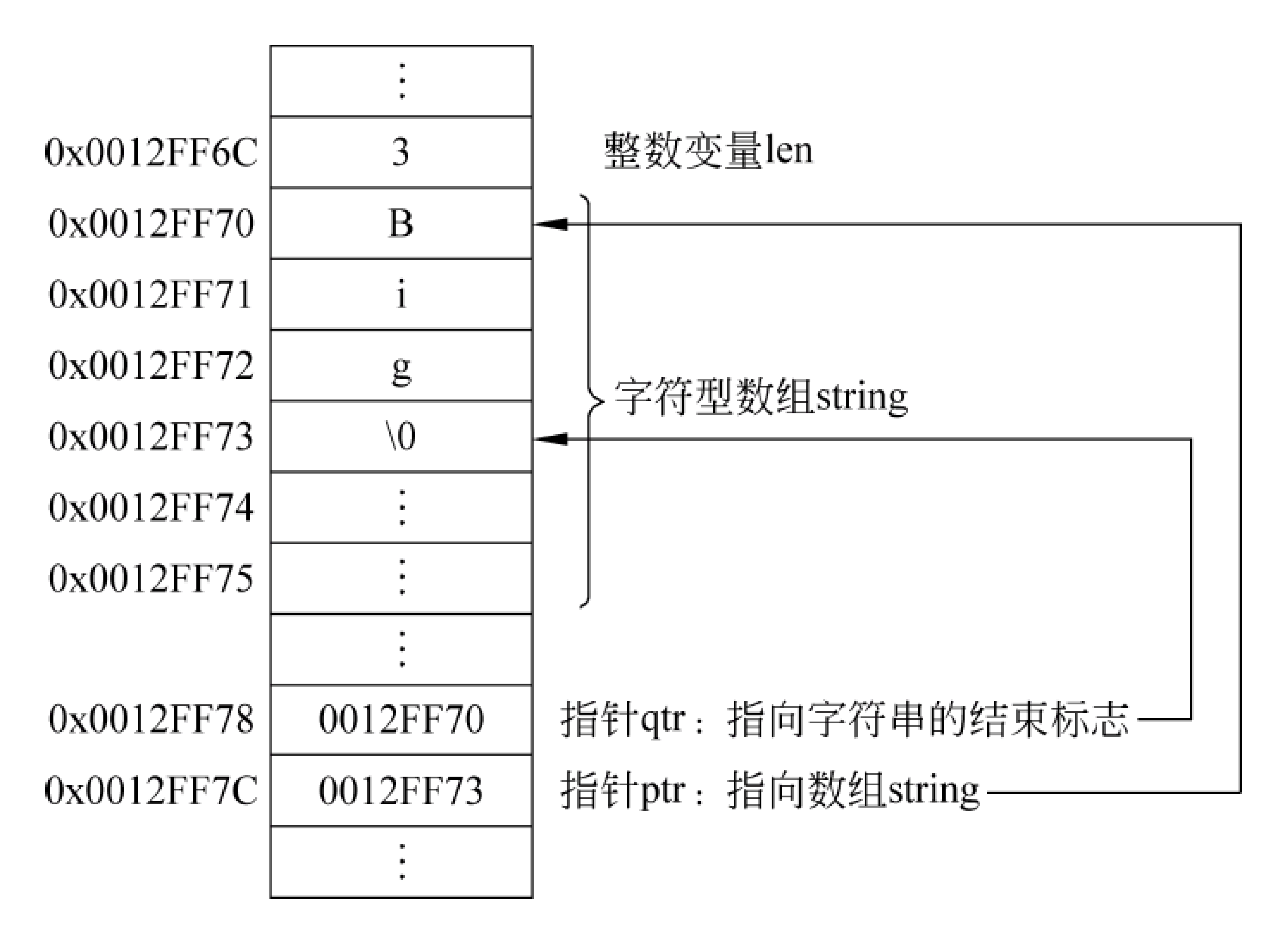


图 6-14　指针的运算

从图 6-14 中可以看出，指针 ptr 指向数组 string 的第一个元素，其内容就是该元素的地址 0x0012FF70。现在，如果执行运算

```
ptr++;
```

即在指针变量 ptr 原来的值上再加 1，使其变为 0x0012FF71，可以看出，这正是数组中第二个元素的地址，也就是说，指针现在改为指向数组中的第二个元素了。如果执行运算

```
ptr+=3;
```

则 ptr 的值由 0x0012FF70 变为 0x0012FF73，即指向数组 string 的第四个元素。由此可以看出，在指针变量上加上一个常数，相当于改变了其中存储的地址值，即改变了指针指向的数组元素。同样，也可以从指针变量存储的地址值上减去一个常数，此时指针向前移动若干个元素。

在图 6-14 中令指针变量 qtr 指向字符串的结束标志'\0'(即数组 string 的第四个元素)。如果求出这两个指针的差：

```
len=qtr-ptr;
```

可以看出，这正是字符串 string 的长度(不含字符串结束符)。

由于在 C 语言中每个变量、数组和函数的具体地址和相对顺序是由连接程序确定的(局部变量是动态分配的)，在编写程序时无法知道其确切地址和相对顺序，所以对于指向单个变量和函数的指针进行这样的运算是没有意义的。但是无论怎样分配，一个数组内的各元素的相对位置总是固定的，所以对数组元素的引用除了使用下标以外，还可以通过

使用指针运算来实现，这是 C 程序设计的一大特点。

【例 6-22】 编写一个字符串复制函数 mystrcpy。

程序：

```
void mystrcpy(char * destin,char * source)
{
    while(* source!=0)              //如果 * source==0 则表示原字符串结束
    {
        * destin= * source;         //复制字符
        source++;                   //source 移向原字符串中的下一个字符
        destin++;                   //destin 移向新字符数组的下一位置
    }
    * destin=0;                     //在新字符串尾部添加一个结束符 0
}
```

分析：因为字符串是以 0 为结束符的字符序列，所以复制字符串也只需复制到 0 为止。由于 C 语言的表达式应用非常灵活，所以这段程序的循环部分也可以写成

```
while((* destin++= * source++) !=0);
```

改写后的程序段只使用了半个语句(while 语句的后半部分被省略了)！但是功能依然不变，甚至新字符串尾部的 0 也已经被复制。C 语言的这种特点是其程序比较精练的原因，受到程序员的偏爱。但是过分的精练也会使程序难以理解，有悖于结构化程序设计的基本原则。

函数 mystrcpy 的功能为将一个字符串的内容复制到另一个字符型数组中。在复制字符串时要注意，一定要保证目标数组确实可以放得下整个字符串。初学者最易犯的一个错误是混淆指针与数组的概念，写出如下的语句：

```
char * string1="This is a sample.";
char * string2;
    ……
mystrcpy(string2,string1);
```

这时确实可以将字符串 string1 中的内容复制到从存放于指针 string2 中的地址开始的一段内存中。但问题是指针 string2 中存放的究竟是谁的地址？由于没有对 string2 赋值，所以它可能指向任何地方，包括已经分配给其他变量、数组甚至函数的区域。向 string2 复制字符串会覆盖这些地方原来的内容，造成各种运行错误，包括突然死机；即使幸而指针 string2 指向一片未被使用的存储区，成功地复制了字符串，但由于没有合法的授权，也不能保证其后程序不再将这片存储区域分配给其他的变量或数组，从而造成刚刚复制的内容又被其他数据覆盖。

解决的方法之一就是可以在变量声明之后添加这样的一段代码：

```
char string [100];
string2=string;
```

这样 string2 就指向了数组 string，而数组一经定义，其首地址就已经确定。

上面的例子使用了字符型的数组，其特点是每个数组元素的大小正好是一个字节。如果使用其他类型的数组，其指针的运算结果有无变化呢？

对于指针变量来说，其运算的基本单位为其指向的数据类型的变量占用的字节数。因此，如果某指针是指向 int 型变量的，由于 int 型的变量占用 4B，所以该指针运算时的基本单位为 4；如果某指针是指向 float 型的，由于 float 型变量的长度为 4B，则该指针运算时的基本单位为 4。

例如，有指针 ptr、qtr 以及整型数组 array：

```
int array[3];
int * ptr=array, * qtr;
qtr=ptr+1;
```

如果 ptr＝0x0012FF74，则 qtr＝0x0012FF78，即在指针变量 ptr 原来的值上加 4，正好使 qtr 指向 array 的第二个数组元素；同样，语句

```
qtr=ptr+3;
```

的结果是变量 qtr 恰好指向数组的末尾。

【例 6-23】 编写一个函数用于将一个 float 型的数组清零（即将其所有元素全部置为 0）。

算法：通过引用下标变量很容易实现数组清零的功能。但在本例中采用指针编写该函数。因为数组元素的类型为 float，所以必须使用指向 float 型的指针。

程序：

```
void clear_array(float * ptr,int len)
{
    float * qtr=ptr+len;
    while(ptr<qtr)
    {
        * ptr=0.0;
        ptr++;
    }
}
```

分析：由程序中可以看出，由于指针运算的基本单位为其指向的类型变量的存储字节数，所以使程序设计变得比较简单。在编写程序时，如果要使指针指向下一个数组元素，不必知道一个数组元素实际占用几个存储单元，只要简单地在指针上加 1 即可。如果要将该函数改为对 double 型的数组清零，只要将指针类型由 float * 改为 double * 即可，其他不必改动。

*6.10.2 指向多维数组的指针

本节以二维数组为例介绍多维数组的指针变量。设有整型二维数组 a[3][4]如下：

1　2　3　4
5　6　7　8
9　10　11　12

它的定义为

```
int a[3][4]={{0,1,2,3},{4,5,6,7},{8,9,10,11}}
```

设数组 a 的首地址为 0x1000，也就是 a[0][0]的地址为 0x1000，a[0][1]的地址是 0x1004，依此类推。C 语言允许把一个二维数组分解为多个一维数组来处理。因此数组 a 可分解为 3 个一维数组，即 a[0]、a[1]、a[2]。每一个一维数组又含有 4 个元素。

例如，a[0]数组含有 a[0][0]、a[0][1]、a[0][2]、a[0][3]共 4 个元素。

从二维数组的角度来看，a 是二维数组名，a 代表整个二维数组的首地址，也是二维数组 0 行的首地址，等于 0x1000。a+1 代表第一行的首地址，也就是 a[1]的地址，等于 0x1010。

a[0]是第一个一维数组的数组名和首地址，因此地址也为 0x1000。*(a+0)或 *a 是与 a[0]等效的，它表示一维数组 a[0]的 0 号元素的首地址，也为 0x1000。&a[0][0]是二维数组 a 的 0 行 0 列元素首地址，同样是 0x1000。因此，a、a[0]、*(a+0)、*a 和 &a[0][0]是等同的。

同理，a+1 是二维数组 1 行的首地址，等于 0x1010。a[1]是第二个一维数组的数组名和首地址，因此也为 0x1010。&a[1][0]是二维数组 a 的 1 行 0 列元素地址，也是 0x1010。因此 a+1、a[1]、*(a+1)和 &a[1][0]是等同的。

此外，&a[i]和 a[i]也是等同的。因为在二维数组中不能把 &a[i]理解为元素 a[i]的地址，不存在元素 a[i]。C 语言规定，它是一种地址计算方法，表示数组 a 第 i 行首地址。由此，a[i]、&a[i]、*(a+i)和 a+i 也都是等同的。

另外，a[0]也可以看成是 a[0]+0，是一维数组 a[0]的 0 号元素的首地址，而 a[0]+1 则是 a[0]的 1 号元素首地址，由此 a[i]+j 则是一维数组 a[i]的 j 号元素首地址，它等于 &a[i][j]。

由 a[i]=*(a+i)得 a[i]+j=*(a+i)+j。由于 *(a+i)+j 是二维数组 a 的 i 行 j 列元素的首地址，所以，该元素的值等于 *(*(a+i)+j)。

把二维数组 a 分解为一维数组 a[0]，a[1]，a[2]之后，设 p 为指向二维数组的指针变量，可定义为

```
int (*p)[4]
```

它表示 p 是一个指针变量，它指向包含 4 个元素的一维数组。若指向第一个一维数组 a[0]，其值等于 a、a[0]或 &a[0][0]等。而 p+i 则指向一维数组 a[i]。从前面的分析可得出，*(p+i)+j 是二维数组 i 行 j 列的元素的地址，而 *(*(p+i)+j)则是 i 行 j 列元素的值。

二维数组指针变量说明的一般形式为

类型说明符 (*指针变量名)[长度]

其中“类型说明符”为所指数组的数据类型。 * 表示其后的变量是指针类型。“长度”表示二维数组分解为多个一维数组时一维数组的长度,也就是二维数组的列数。应注意“(* 指针变量名)”两边的括号不可少,如果缺少括号则表示是指针数组(本章后面介绍),意义就完全不同了。

6.10.3 指针数组

指针数组也是数组,但它和一般数组不同,其数组元素不是一般的数据类型,而是指针,即内存单元的地址。这些指针必须指向同一种数据类型的变量。指针数组的声明方式和普通数组的声明方式类似,在数组名后加上维长说明即可。

声明一维指针数组的语法形式为

数据类型 * 数组名[常量表达式];

其中“常量表达式”指出数组元素的个数,“数据类型”确定每个元素指针的类型,“数组名”是指针数组的名称,同时也是这个数组的首地址。例如,声明一个一维指针数组,其中包括 10 个数组元素,均为指向字符类型的指针:

```
char * ptr[10];
```

当然也可以声明二维以至多维指针数组,例如:

```
int * index[10][2];
```

【例 6-24】 编写一个查词典的函数。词典以词条为单位,词条的格式为

knowledge: n. 知识,学问,认识

每个词条使用一个字符型数组存放,整个词典使用一个指向字符类型的指针数组表示,其中每个指针指向一个词条。其拓扑结构如图 6-15 所示。

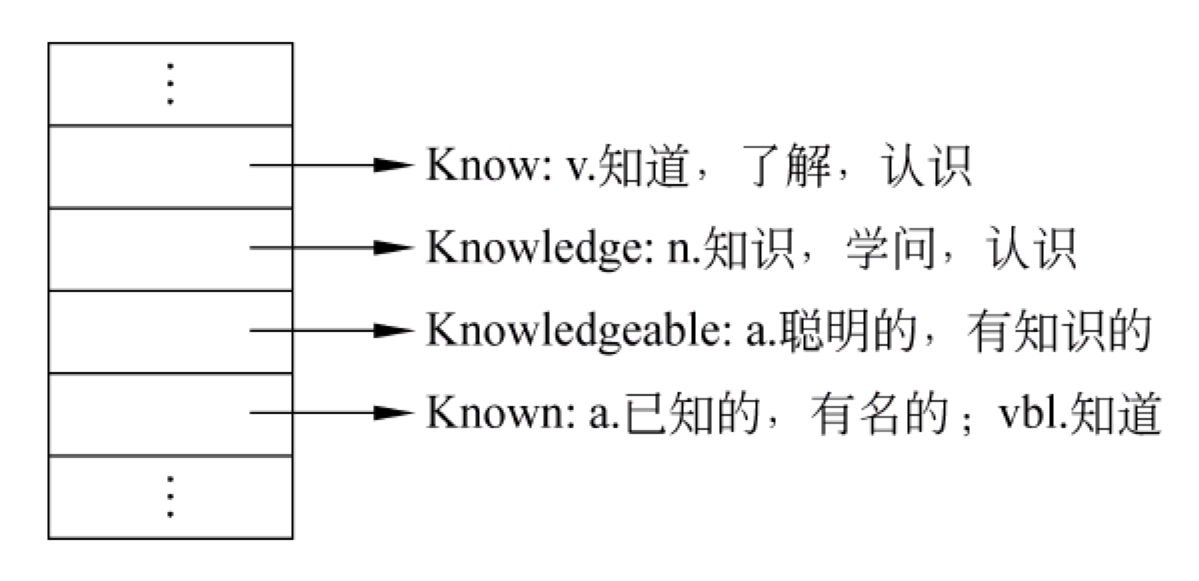


图 6-15 词典的拓扑结构

算法:在一个线性表(如数组)中进行查找是程序设计的经典题目,不同的查找算法的效率(可以从查找速度和需用的存储单元数目两个方面考查)相差很大。对于无序表来说,一般只能采用顺序查找法,其速度最慢。如果设表中元素数目为 n 个,则平均查找长度(查找长度是为了找到指定元素所进行的比较次数)为 $n/2$。如果所查找的内容不在表中,则必须将整个表中所有元素都浏览一遍后才能知道,即最大查找长度为 n。

对于有序表来说，则可以采用二分法进行查找。二分法查找的算法为：首先将关键字（即查找内容）与位于表正中的元素进行比较，此时不外 4 种情况（设表按升序排列，下同）：

（1）关键字等于该中点元素：查找成功，结束查找过程。

（2）关键字小于该中点元素：说明关键字在表的前半部分，因此可以将表的前半部分作为一个新表（表长小于原来表的一半），继续应用本算法在新表中进行查找。

（3）关键字大于该中点元素：说明关键字在表的后半部分，因此可以将表的后半部分作为一个新表，继续应用本算法在新表中进行查找。

（4）表长已等于 0：说明表中没有要查找的内容，查找失败。

很容易看出，这是一个递归算法，其最大查找长度为 $O(\log_2 n)$。在 n 比较大的场合，二分法查找明显优于顺序查找。例如，若表长为 1024，则顺序查找的平均查找长度为 512，最大查找长度为 1024；而二分法的最大查找长度不过为 10。表越长，二分法查找的优点越明显。

二分法查找的缺点是事先要将表排序。排序的代价很高，因此为了一两次查找就对表进行排序是不值得的。只有在查找次数频繁，而表中内容不经常变动时采用排序加二分法查找才是划算的。

词典的排列是有序的，而且其长度通常都比较大，内容相对固定，采用二分法查找正合适。设词典存放在长度为 n 的一维表 dict 中，表中元素为指向词条字符串的指针；待查单词为 word；并有 3 个工作变量 low、high 和 mid，分别用于记录待查表的低端、高端和中点元素的下标。使用伪代码将上述二分法算法细化如下：

```
low = 0;                                //设置工作变量的值
high= n- 1;
do
{
    mid = (low+ high)/2;                //算出表中点元素的下标
    if(word 等于 dict[mid])
        则 dict[mid]即为所要查找的词条,查找成功,结束查找;
    else if(word 在 dict[mid]之前)
        high= mid- 1;                   //把表缩小为原表的前一半
    else if(word 在 dict[mid]之后)
        low= mid+ 1;                    //把表缩小为原表的后一半
}while(表长度大于 0);
//如果循环正常结束,说明查找失败
```

程序：

```
//二分法查词典
char * search_word(char * word,char * dict[],int n)
{
    int low= 0,high= n- 1,mid,searchpos,wordlen= strlen(word);
```

```
    do
    {
        mid= (low+ high)/2;                    //算出表中点元素的下标
        searchpos= strnicmp(word,dict[mid],wordlen);                    //利用字符串比较库函数
        if(searchpos= = 0)                     //查找成功
            return dict[mid];
        else if(searchpos< 0)
                high= mid- 1;                  //把表缩小为原表的前一半
            else
                low = mid+ 1;                  //把表缩小为原表的后一半
    }while(high> low);
    return NULL;                               //查找失败
}
```

分析：如果查找的关键字是数值，则可以简单地使用等于、大于和小于等比较运算符判断进一步查找的路线。但对于字符串查找来说，问题要稍微复杂些。字符串的大小是按ASCII码字典序确定的，排在前面的单词小，排在后面的单词大。库函数strcmp正好可以用来比较两个字符串。但在本例的词典查找任务中还有几个问题需要认真考虑。一是在真正的词典中，大小写字母是排在一起的，而在ASCII码表中所有的大写字母排在所有的小写字母之前。考虑到这种差别，如果不准备打乱通常词典中的词序重新排列，就必须在比较时忽略大、小写字母的差别。二是在上述查找词典算法中是将待查单词和词典中的词条直接进行比较，如果直接使用strcmp之类的比较函数，就会发现后者的长度大于前者的长度，从而认为这两个字符串不相等。解决的办法之一是在比较时只比较待查单词和词条的前几个字符，如果相等就认为比较成功。符合上述要求的字符串比较库函数为strnicmp。

6.11 指针与函数

6.11.1 指针作为函数的参数

在C语言中，函数的参数不仅可以是基本数据类型的变量、对象名、数组名或函数名，而且可以是指针。当以指针作为形参时，在函数调用过程中实参将值传递给形参，也就是使实参和形参指针变量指向同一内存地址。这样对形参指针所指变量值的改变也同样影响着实参指针所指向的变量的值，也就是说，通过使实参与形参指针指向共同的内存空间，达到了参数双向传递的目的。

6.11.2 返回指针的函数

一般来说，函数可以用返回值的形式为调用程序提供一个计算结果。在前面的各章

中出现的函数返回值类型大都是 int、double 之类的简单类型。其实,也可以将一个地址(如变量、数组和函数的地址,指针变量的值等)作为函数的返回值。在说明返回值为地址的函数时,要使用指针类型说明符,例如:

```
char * strchr(char * string,int c);
char * strstr(char * string1,char * string2);
```

这是两个用于字符串处理的库函数,其返回值均为地址。前者的功能为在字符串 string 中查找字符 c,如果字符串 string 中有字符 c 出现,则返回字符 c 的地址,否则返回 NULL。后者的功能为在字符串 string1 中查找子字符串 string2,如果字符串 string1 中包含子字符串 string2,则返回 string2 在 string1 中的地址(即 string2 中第一个字符的地址),否则返回空指针值 NULL。

【例 6-25】 将表示月份的数值(1～12)转换成对应的英文月份名称。

算法:首先说明一个字符串数组 month,用来存放月份的英文名称。在转换时只需按下标值返回一个字符串的地址即可。

程序:

```
//将月份数值转换为相应的英文名称
char * month_name(int n)
{
    static char * month[]=
    {
        "Illegal month",        //月份值错
        "January",              //一月
        "February",             //二月
        "March",                //三月
        "April",                //四月
        "May",                  //五月
        "June",                 //六月
        "July",                 //七月
        "August",               //八月
        "September",            //九月
        "October",              //十月
        "November",             //十一月
        "December"              //十二月
    };
    return (n>=1 && n<=12)?month[n]:month[0];
}
```

分析:

(1) 变量 month 是一个字符型指针数组,其初值必须是用双引号括起来的字符串。数组中各个指针变量的值分别是 13 个字符串的首地址(也是字符串中第 1 个字符的地址),该数组中的第一个字符串表示输入错误的月份值时的提示。

(2) 数组元素 month[i]是 month 中第 i 个字符型指针(0≤i<13),它指向了第 i 个字符串。即 month[i]存入的是第 i 个字符串的首地址。若想从第 i 个字符串中第 2 个字符开始输出部分子串。可以通过移动指针来实现:

```
month[i]+1;                    //指针加 1,指向字符串的第 2个字符
```

(3) 在处理多个长度不同的字符串时,使用指针数组要比二维数组节省空间。

*6.11.3 指向函数的指针

程序只有装入内存以后才能运行。函数本身作为一段程序,其代码也在内存中占有一片存储区域,这些代码中的第一个代码所在的内存地址称为首地址。首地址是函数的入口地址。主函数在调用子函数时,就是让程序转移到子函数的入口地址开始执行。

所谓指向函数的指针,就是指针的值为该函数的入口地址。

指向函数的指针变量的说明格式为

<函数返回值类型说明符>(*<指针变量名>)(<参数说明表>);

例如:

```
int(*p)();                     //p为指向返回值为整型的函数的指针
float(*q)(float,int);          //q为指向返回值为浮点型函数的指针
```

第 5 章讲过,函数名与数组名类似,表示该函数的入口地址,因此可以直接把函数名赋给指向函数的指针变量。

注意,在说明指向函数的指针变量时,指针变量名前后的圆括号不能缺少。试比较

```
int *func();                   //返回地址的函数
int (*func)();                 //指向函数的指针
```

前者说明了一个函数,其返回值为指向整型的指针;而后者说明了一个指向返回值为整型的函数的指针变量,意义完全不同。

如果已经将某函数的地址赋给一个指向函数的指针变量,就可通过该指针变量调用函数。例如:

```
double (*func)(double)=sin;    //说明一个指向函数的指针
double y,x;                    //说明两个双精度类型的变量
x=…;                           //计算自变量 x的值
y=(*func)(x);                  //通过指针调用库函数求 x的正弦值
```

【例 6-26】 通用的一元数值积分函数。

算法:采用梯形积分公式,将被积函数作为积分函数的参数设计。

程序:

```
//用梯形积分法求解定积分的通用积分函数
```

```
double integral(double a,double b,double (* fun)(double),int n)
{
    double h = (b- a)/n;
    double sum= ((* fun)(a)+ (* fun)(b))/2;
    int i;
    for(i= 1;i< n;i+ + )
        sum += (* fun)(a+ i* h);
    sum * =h;
    return sum;
}
```

分析：这个程序本身很简单。为了能够计算不同被积函数的定积分，将被积函数也设计为一个参数，实际上是将被积函数的调用地址传递给积分函数。在上述积分函数内部，通过指向函数的指针调用被积函数来计算相应的函数值。例如要计算

$$\int_0^1 \sin x \mathrm{d}x$$

可以这样调用函数 integral：

```
double s;
s= integral(0.0,1.0,sin,1000);              //积分区间等分为 1000 份
```

6.12　动态存储分配

一般来说，程序中使用的变量和数组的类型、数目和大小是在编写程序时由程序员确定下来的，因此在程序运行时这些数据占用的存储空间也是一定的。这种存储分配方法被称为静态存储分配。静态存储分配的缺点是程序无法在运行时根据具体情况（如用户的输入）灵活调整存储分配情况。例如，无法根据用户的输入决定程序能够处理的矩阵的规模。

动态存储分配机制为克服这种不便提供了手段。常用的内存管理函数有以下 3 个（需要头文件 stdlib. h）。

1. 分配内存空间函数 malloc

调用形式：

(类型说明符 *)malloc(size)

功能：在内存的动态存储区中分配一块长度为 size 字节的连续区域。函数的返回值为该区域的首地址。“类型说明符”表示把该区域用于何种数据类型。“(类型说明符 *)”表示把返回值强制转换为该类型指针。size 是一个无符号数。

例如：

```
pc= (char * )malloc(100);
```

表示分配 100B 的内存空间，并强制转换为字符数组类型，函数的返回值为指向该字符数组的指针，把该指针赋予指针变量 pc。

2. 分配内存空间函数 calloc

calloc 也用于分配内存空间。

调用形式：

```
(类型说明符*)calloc(n,size)
```

功能：在内存动态存储区中分配 n 块长度为 size 字节的连续区域。函数的返回值为该区域的首地址。“(类型说明符*)”用于强制类型转换。calloc 函数与 malloc 函数的区别在于一次可以分配 n 块区域。还有一个重要的区别就是 calloc 会将分配的内存初始化为 0。

例如：

```
ps=(struet stu*)calloc(2,sizeof(struct stu));
```

其中的 sizeof(struct stu)是求 stu 的结构长度。因此该语句的意思是：按 stu 的长度分配 2 块连续区域，强制转换为 stu 类型，并把其首地址赋予指针变量 ps。

3. 释放内存空间函数 free

调用形式：

```
free(void* ptr);
```

功能：释放 ptr 所指向的一块内存空间，ptr 是一个任意类型的指针变量，它指向被释放区域的首地址。被释放区应是由 malloc 或 calloc 函数所分配的区域。

动态存储分配的变量和数组通过指针来访问。例如：

```
int x,*ptr=(int*)malloc(5*sizeof(int));
*ptr=5;
x=*ptr;
```

【例 6-27】 利用动态数组来求斐波那契数列的前 n 项。

程序：

```
//用动态数组来求斐波那契数列的前 n项
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    printf("Please input n=? ");
    scanf("%d",&n);
    int *p = (int*)malloc(sizeof(n)*(n+1));
```

```
    //如果没有申请到内存或数据输入有误,则返回
    if ( p==NULL || n<=0 )
    {
        printf("Error!\n");
        return -1;
    }
    p[0]=0;
    p[1]=1;
    printf("%d\n",p[0]);
    printf("%d\n",p[1]);
    for(int i=2;i<=n;i++)
    {
        p[i]=p[i-2]+p[i-1];
        printf("%d\n",p[i]);
    }
    free(p);                                //释放数组空间
    return 0;
}
```

分析：本章习题 1 要求用静态数组计算斐波那契数列的前 n 项的问题。由于不知道数组的元素个数,所以在处理时只好按最多的元素计算,而现在使用动态数组,就能够按实际所需进行存储单元的分配,避免不必要的浪费。在使用完存储单元后,还可以释放所占用的存储单元。

使用动态存储分配时要注意几个问题：一是要确认分配成功后才能使用,否则可能造成严重后果;二是在分配成功后不宜变动指针的值,否则在释放这片存储区域时会引起系统内存管理混乱;三是动态分配的存储空间不会自动释放,只能通过 free 释放,因此要注意适时释放动态分配的存储空间。例如：

```
void func()
{
    int *ptr=(int*)malloc(20);
    ……
}
```

由于 ptr 是局部变量,在退出 func 函数后自动失效。如果在函数 func 中没有及时释放动态分配的存储单元,则在退出 func 函数后再也找不到这些单元的地址了。

*6.13　指向指针的指针

指针也是变量,当然也有地址。其地址也可以使用地址运算符 & 求出。那么,指针的地址可否存储在某种变量中呢?答案是肯定的。能够存放指针地址的变量当然也是指针,是“指向指针的指针”。指向指针的指针的说明方法为

<数据类型>**<指针变量名>;

例如:

```
int x=2;
int * xp, * * xpp;
xp = &x;
xpp= &xp;
```

其内存分配情况见图 6-16。

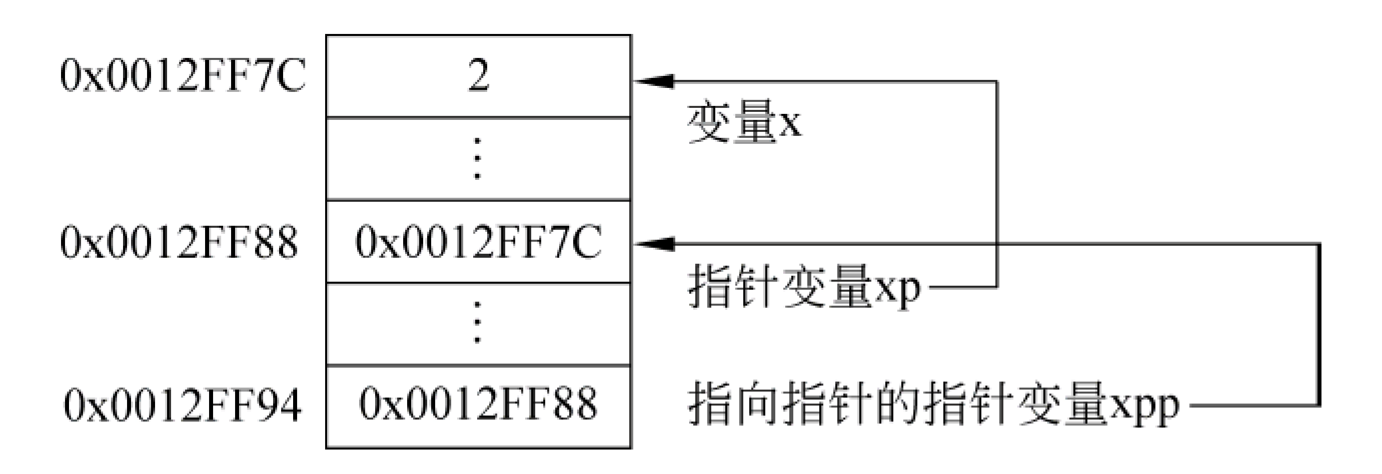


图 6-16　指向指针的指针内存分配示意图

对于指向指针的指针来说,用 * 运算符可以求出其存储的地址的内容,因为该内容仍然是一个地址,所以可以再次应用 * 运算符,求该地址存储的内容。对于上面的例子,* xpp 就是 xp 的内容,即 x 的地址,而 * * xpp 则是 x 的值。

指向指针的指针有以下几个用途。

(1) 可以用于指针数组的处理。例如,在例 6-24 中的词典查找程序中,使用了一个指针数组存放词典的条目。该数组的元素是指向字符类型变量的指针,所以可以说明一个指向指针的指针 nextword 存放词典 dict 的某个元素的地址:

```
char **nextword= &(dict[i]);
```

则 * nextword 为词典 dict 的某一元素的内容,也就是某词条字符串的地址。而通过

```
nextword++;
```

这样的运算,可以遍历 dict 的各个元素,对于设计查找同源词的程序相当有用。

(2) 作为函数的参数。要修改作为参数的变量的值,必须使用指针;那么要修改作为参数的指针的值,就必须使用指向指针的指针。

(3) 作为函数的返回值。

既然"指向指针的指针"还是一个指针变量,当然也有地址,那么有没有"指向'指向指针的指针'的指针"呢?

从理论上说,如果将指向指针的指针称为二级指针,则"指向指向指针的指针"可以称为三级指针,在此基础上还可以定义四级指针、五级指针等多级指针。只是在一般的编程实践中,难得遇到使用多于二级指针的情况。加之多于二级的多级指针应用过于复杂,编程、调试都有困难,所以应用不多。

6.14　结构体与指针

因为结构体也是存储在内存单元中的，所以它也是有地址的，同样也可以用取地址运算符 & 来得到结构体类型变量的地址。这也就是说可以用指针来访问它，即可以声明指向结构体的指针。

声明指向结构体的指针方法与前面所讲的方法相同。例如，对于以下结构体类型：

```
struct StudentType
{
    char id[10];            //学号
    double score[5];        //五门课程成绩
    double GPA;             //平均分
};
```

可以有

```
struct StudentType xjtuStudent;
struct StudentType * ptr=&xjtuStudent;
```

这里就定义了一个指向结构体的指针 ptr，并将 xjtuStudent 的地址赋给它，如图 6-17 所示。

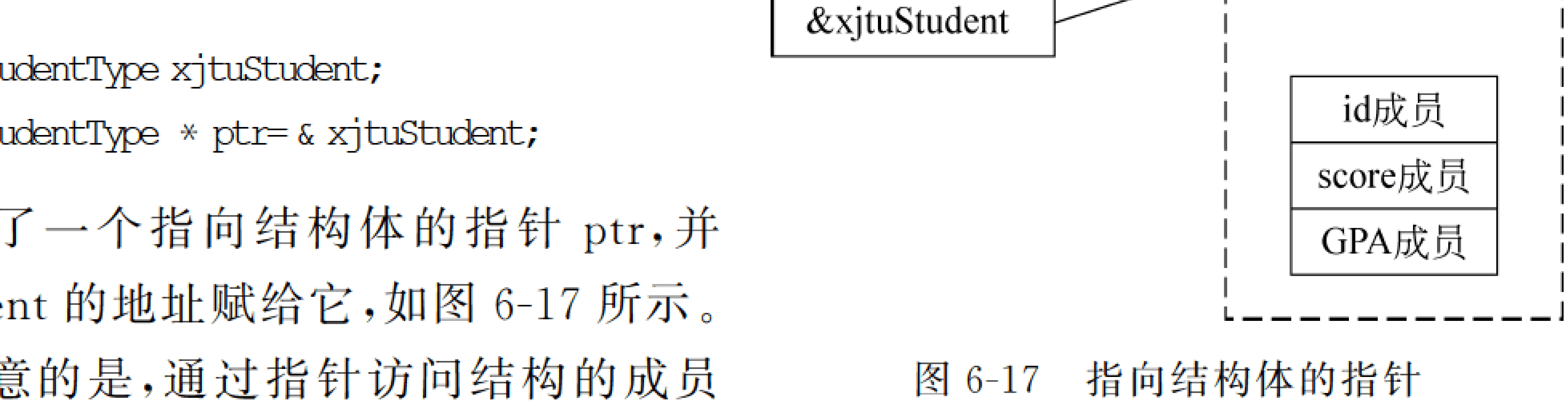


图 6-17　指向结构体的指针

但要注意的是，通过指针访问结构的成员要用箭头操作符“—>”，例如：

```
ptr->score[1]=90;
```

6.15　指针的初始化

声明一个指针变量后，如果没有对它赋初值，则它的值（即它所指向的内存位置）是不确定的，这时直接对指针指向的内存写入数据是极其危险的。为避免上述错误，一般需要在声明时对指针变量进行初始化。

1. 指针变量的初始化

定义指针变量的同时，赋予该指针变量初值，其一般形式为

数据类型标识符 * 指针变量名=初始地址值;

例如：

```
int i;
int * ptr=&i;
```

上面第一个语句定义一个整型变量 i，系统为 i 分配一个内存空间（因而相应有一个地址）；第二个语句定义 ptr 为指针变量，同时将 i 的地址赋予指针变量 ptr 作为初值。这两个语句的顺序不能颠倒。

指向字符类型的指针可以用字符串进行初始化，例如：

```
char * string="Hello,Visual C++!";
```

实际上，在编程实践中，程序员经常使用如下的初始化语句：

```
int * ptr=NULL;
```

这里指针 ptr 被初始化为 NULL，即空指针。NULL 是一个在头文件＜stdio. h＞中定义的符号化常量，将指针初始化为 NULL 就等于将指针初始化为 0。值为 NULL 的指针不指向任何变量。

在定义指针时将指针初始化为 NULL 是一个很好的编程习惯，这样做可以防止该指针变量指向某一个未知的内存区域而产生难以预料的错误。

2. 指针数组的初始化

指针数组在声明的同时可以进行初始化。例如：

```
char * func_namelist[] =
{
    "strcat","strchr","strcmp","strcpy","strlwr","strstr","strupr"
};
```

6.16 void 和 const 类型的指针

可以说明指向 void 类型的指针，但其含义有所不同。指向 void 类型的指针是通用型的指针，可以指向任何类型的变量。可以直接对 void 型指针赋值或将其与 NULL 作比较，但是在求指针的对象变量的内容或者进行指针运算之前必须对其进行强制类型转换。例如：

```
int x,y;
void * ptr;
ptr=&x;                //任何类型变量的地址均可存入指向 void类型的指针
y = * ((int * )ptr);   //通过 void型指针求值时要用强制类型转换
```

用关键字 const 修饰一个指针时，根据其位置的不同有不同的含义。例如：

```
const char * ptr="Point to constant string";
```

表示定义了一个指针 ptr，它指向一个常数字符串。因此，运算

```
 * ptr='Q';
```

是非法的，因为该字符串为常量。但指针 ptr 本身为变量，可以修改。例如：

```
ptr++;
```

合法。而

```
char * const qtr="A constant pointer";
```

定义了一个常指针。在这种情况下，指针本身不能修改，但其指向的对象并非常量，允许修改。

实际上，修饰符 const 多用于修饰函数的指针或引用参数，以防止在编程中无意识地改变其值。例如：

```
double func1(const double * x);
```

如果在编写函数 func1 的代码时改变了指针对象或引用对象的值，则会引起编译错误。

6.17 指针应用示例

【例 6-28】 编写一个字符串比较函数，仅比较两个字符串的前面若干个字符，且在比较时不区分大小写字母。

算法：例 6-19 介绍的字符串比较函数 mystrcmp 的编写方法是通过下标对数组进行操作的。实际上使用指针处理这类操作会更加方便。为了达到在比较时不区分大小写字母的目的，可以使用库函数 toupper，其原型为

```
int toupper(int c);          //include<ctype.h>
```

其中参数 c 为待转换的 ASCII 代码，如果 c 是一个小写字母，则该函数返回与其对应的大写字母。

程序：

```
//不区分大小写字母的部分字符串比较
int mystrnicmp(char * str1,char * str2,int n)
{
    while(toupper(* str1)==toupper(* str2) && * str1 && * str2 && n>0)
    {
        str1++;
        str2++;
        n--;
    }
    return * str1- * str2;
}
```

分析：该函数的核心是 while 语句中的条件。只有以下 3 个条件同时满足，循环才能

继续执行。

(1) 不计大小写字母的区别,两个字符串中对应位置上的字符相等。

(2) 两个字符串均未结束。

(3) 已经比较过的字符尚未达到指定的数目。

因此循环结束时的状况必然是以上条件中有一条或多条已不满足。通常,应对这些条件逐一进行判断,分别处理。但本函数是一个特例,无论因为什么原因结束循环,均返回表达式 * str2－ * str1 的值。如果返回值为 0,表示两个字符串相同(在忽略大小写字母的区别和仅比较两个字符串的前 n 个字符的前提下,下同);如果返回值大于 0,表示字符串 str2 大于 str1;否则表示 str2 小于 str1。

【例 6-29】 删除字符串中的子串。输入一个字符串 a,再输入一个字符串 b,然后删除字符串 a 中出现的所有子串 b,最后输出 a。

分析:设计一个函数 char * f(char * p,char * s),功能是在 p 指向的字符串中查找并删除 s 指向的子串。若找到多个子串则必须全部删除,删除完成后返回结果字符串的首地址。函数 main 输入字符串与子串,调用函数 f 后输出结果字符串。首先考虑如何在字符串 p 中删除子串 s。由于 p 可能包含多个子串 s,而且这些子串不一定是紧挨着的,若先找出全部子串再一起删除这些子串会比较麻烦,所以不妨采取找到一个删除一个的办法。当在 p 中找到一个子串 s 后,如何删除它?比较简单的做法是把 p 中子串 s 后面的所有字符全部向前移动到 p 中子串 s 首字符的位置。

程序:

```
#include <stdio.h>
#include <string.h>
char * del(char * p,char * s)
{
    int slen,tlen,i;
    char * t=p;
    slen=strlen(s);
    while(strlen(t) >=slen)
    {
        for(i=0;i<slen;i++)                //比较 t 所指字符串与子串 s
        {
            if(t[i] !=s[i])
                break;
        }
        if(i <slen)                        //若 t 所指字符串不等于子串 s
        {
            t++;                           //使 t 指向下一个字符
            continue;                      //并继续下一次循环
        }
        tlen=strlen(t);
        for(i=0;i<tlen-slen+1;i++)         //删除找到的一个子串
```

```
        {  //把 t 所指字符串中子串 s 后面的所有字符 (包含结束符'\0')向前移动 slen 个字符
           t[i]=t[i+slen];
        }
    }
    return p;
}
void main()
{
    char a[100],b[100];
    gets(a);                        //输入目标字符串
    gets(b);                        //输入子串
    puts(f(a,b));                   //在目标串 a 中删除子串 b
}
```

【例 6-30】 输入若干字符串,按 A～Z 顺序排序后输出。设计一个函数 void f(char **p,int n),其中 p 指向某个指针数组的首元素,函数的功能是对指针数组各元素所指向的 n 个字符串按 A～Z 顺序排序,要求排序过程中不能对字符串本身作交换而是对各个字符指针作交换。

分析:函数 f 的第 1 个参数为指针的指针,实际上该参数也可以定义为 char * p[]。因为,当用指针数组作函数的实参时,函数的形参也可以写成与实参一样的指针数组形式,但是要注意 char * p[]中的 p 本质上并非数组,而是指针变量,其类型为 char * * p。在函数 f 中,可以用 strcmp 对两个字符串进行比较。若 strcmp 返回负数,则第 1 个字符串小于第 2 个字符串;若 strcmp 返回正数,则第 1 个字符串大于第 2 个字符串;若 strcmp 返回 0,则两个字符串相等。例如,strcmp("brown","quick")会返回一个负数,因为'b'<'q'。排序方法可以使用简单选择排序算法(参看 7.5 节的排序算法)。

程序:

```
#include <stdio.h>
#include <string.h>
void f(char * * p,int n)
{
    int i,j,min;
    char * t;
    for(i=0;i<n-1;i++)
    {
        min=i;
        for(j=i+1;j<=n-1;j++)
        {
            if(strcmp(p[j],p[min])<0)
                min=j;
        }
        t=p[i];
        p[i]=p[min];
```

```
            p[min]=t;
        }
}
void main()
{
        int i;
        char a[4][100],*b[4];
        for(i=0;i<4;i++)
        {
            gets(a[i]);
            b[i]=a[i];
        }
        f(b,4);
        for(i=0;i<4;i++)
            puts(b[i]);
}
```

6.18 预处理命令

在前面各章中,已多次使用过以#开头的预处理命令。如包含命令#include、宏定义命令#define等。在源程序中这些命令都放在函数之外,而且一般都放在源文件的前面,它们称为预处理部分。所谓预处理是指在进行编译之前所做的工作。预处理是C语言的一个重要功能,它由预处理程序负责完成。当对一个源文件进行编译时,系统将自动引用预处理程序对源程序中的预处理部分作处理,处理完毕自动进入对源程序的编译。

C语言提供了多种预处理功能,如宏定义、文件包含、条件编译等。合理地使用预处理功能编写的程序便于阅读、修改、移植和调试,也有利于模块化程序设计。

6.18.1 无参数宏

在C语言源程序中允许用一个标识符来表示一个字符串,称为宏。被定义为宏的标识符称为宏名。在编译预处理时,对程序中所有出现的宏名,都用宏定义中的字符串去代换,这称为宏代换或宏展开。宏定义是由源程序中的宏定义命令完成的。宏代换是由预处理程序自动完成的。在C语言中,宏分为有参数和无参数两种。无参宏的宏名后不带参数。其定义的一般形式为

#define 标识符 字符串

其中的#表示这是一条预处理命令,凡是以#开头的均为预处理命令。define为宏定义命令。“标识符”为所定义的宏名。“字符串”可以是常数、表达式、格式串等。

在前面介绍过的符号常量的定义就是一种无参宏定义。此外,常对程序中反复使用

的表达式进行宏定义。例如：

```
#define M(y* y+3* y)
```

它的作用是指定标识符 M 来代替表达式(y * y+3 * y)。在编写源程序时，所有的(y * y+3 * y)都可由 M 代替，而对源程序作编译时，将先由预处理程序进行宏代换，即用(y * y+3 * y)表达式去置换所有的宏名 M，然后再进行编译。

对于宏定义还要说明以下几点：

(1) 宏定义是用宏名来表示一个字符串，在宏展开时又以该字符串取代宏名，这只是一种简单的代换，字符串中可以含任何字符，可以是常数，也可以是表达式，预处理程序对它不作任何检查。如果有错误，只能在编译已被宏展开后的源程序时发现。

(2) 宏定义不是说明或语句，在行末不必加分号，如果加上分号则连分号也一起置换。

(3) 宏定义必须写在函数之外，其作用域为从宏定义命令起到源程序结束。如果要终止其作用域可使用 #undef 命令。例如：

```
#define PI 3.14159
main()
{
    ……
}
#undef PI
f1()
{
    ……
}
```

表示 PI 只在 main 函数中有效，在 f1 中无效。

(4) 宏名在源程序中若用引号括起来，则预处理程序不对其作宏代换。

(5) 宏定义允许嵌套，在宏定义的字符串中可以使用已经定义的宏名。在宏展开时由预处理程序层层代换。例如：

```
#define PI 3.1415926
#define S PI* y* y                    /* PI 是已定义的宏名 */
```

对语句

```
printf("%f",S);
```

在宏代换后变为

```
printf("%f",3.1415926* y* y);
```

(6) 习惯上宏名用大写字母表示，以便于与变量区别，但也允许用小写字母。

(7) 可用宏定义表示数据类型，使书写方便。例如：

```
#define STU struct stu
```

在程序中可用 STU 作变量说明：

```
STU body[5], * p;
```

应注意用宏定义表示数据类型和用 typedef 定义数据说明符的区别。宏定义只是简单的字符串代换，是在预处理时完成的；而 typedef 是在编译时处理的，它不是作简单的代换，而是对类型说明符重新命名，被命名的标识符具有类型定义说明的功能。

6.18.2 带参宏定义

C 语言允许宏带有参数。在宏定义中的参数称为形式参数，在宏调用中的参数称为实际参数。对带参数的宏，在调用中，不仅要宏展开，而且要用实参去代换形参。带参宏定义的一般形式为

```
#define 宏名(形参表) 字符串
```

在字符串中含有各个形参。

带参宏调用的一般形式为：

```
宏名(实参表);
```

例如：

```
#define M(y) y* y+3* y            /* 宏定义 */
……
k=M(5);                           /* 宏调用 */
……
```

在宏调用时，用实参 5 去代替形参 y，经预处理宏展开后的语句为

```
k=5* 5+3* 5
```

对于带参的宏定义有以下问题需要说明：

(1) 带参宏定义中，宏名和形参表之间不能有空格出现。例如：

```
#define MAX(a,b) (a>b)?a:b
```

写为

```
#define MAX (a,b) (a>b)?a:b
```

将被认为是无参宏定义，宏名 MAX 代表字符串(a,b)(a>b)?a:b。

(2) 在带参宏定义中，形式参数不分配内存单元，因此不必作类型定义。而宏调用中的实参有具体的值。要用它们去代换形参，因此必须作类型说明。这是与函数中的情况不同的。在函数中，形参和实参是两个不同的量，各有自己的作用域，调用时要把实参值赋予形参，进行“值传递”。而在带参宏中，只是符号代换，不存在值传递的问题。

(3) 在宏定义中的形参是标识符，而宏调用中的实参可以是表达式。宏调用与函数

的调用不同的是，函数调用时要把实参表达式的值求出来再赋予形参，而宏代换中对实参表达式不作计算，直接照原样代换。

（4）在宏定义中，字符串内的形参通常要用括号括起来以避免出错。

6.18.3 文件包含

文件包含是C语言预处理程序的另一个重要功能。文件包含命令行的一般形式为

```
#include"文件名"
```

在前面已多次用此命令包含过库函数的头文件。例如：

```
#include"stdio.h"
#include"math.h"
```

文件包含命令的功能是把指定的文件插入该命令行位置取代该命令行，从而把指定的文件和当前的源程序文件连成一个源文件。在程序设计中，文件包含是很有用的。一个大的程序可以分为多个模块，由多个程序员分别编程。有些公用的符号常量或宏定义等可单独组成一个文件，在其他文件的开头用包含命令包含该文件即可使用。这样，可避免在每个文件开头都书写那些公用量，从而节省时间，并减少出错。对文件包含命令还要说明以下几点：

（1）包含命令中的文件名可以用双引号括起来，也可以用尖括号括起来。例如，以下写法都是允许的：

```
#include"stdio.h"
#include<math.h>
```

但是这两种形式是有区别的：使用尖括号表示在包含文件目录中去查找（包含目录是由用户在设置环境时设置的），而不在源文件目录去查找；使用双引号则表示首先在当前的源文件目录中查找，若未找到才到包含目录中去查找。用户编程时可根据自己文件所在的目录来选择某一种命令形式。

（2）一个include命令只能指定一个被包含文件，若有多个文件要包含，则需用多个include命令。

（3）文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。

*6.18.4 条件编译

预处理程序提供了条件编译的功能。可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件。这对于程序的移植和调试是很有用的。

条件编译有3种形式，下面分别介绍。

第一种形式如下：

```
#ifdef 标识符
```

```
    程序段 1
#else
    程序段 2
#endif
```

它的功能是，如果标识符已被#define 命令定义过，则对程序段 1 进行编译，否则对程序段 2 进行编译。如果没有程序段 2（它为空），本格式中的#else 可以没有，即可以写为

```
#ifdef 标识符
    程序段
#endif
```

第二种形式如下：

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

与第一种形式的区别是将 ifdef 改为 ifndef。它的功能是，如果标识符未被#define 命令定义过，则对程序段 1 进行编译，否则对程序段 2 进行编译。这与第一种形式的功能相反。

第三种形式如下：

```
#if 常量表达式
    程序段 1
#else
    程序段 2
#endif
```

它的功能是，如果常量表达式的值为真（非 0），则对程序段 1 进行编译，否则对程序段 2 进行编译，因此可以使程序在不同条件下完成不同的功能。

习　　题

1. 使用数组来求斐波那契数列的第 n 项和前 n 项之和。
2. 编写程序，将 4 阶方阵转置，如下所示。

$$\begin{bmatrix} 4 & 6 & 8 & 9 \\ 2 & 7 & 4 & 5 \\ 3 & 8 & 16 & 15 \\ 1 & 5 & 7 & 11 \end{bmatrix} \Rightarrow \begin{bmatrix} 4 & 2 & 3 & 1 \\ 6 & 7 & 8 & 5 \\ 8 & 4 & 16 & 7 \\ 9 & 5 & 15 & 11 \end{bmatrix}$$

转置前方阵$\boldsymbol{A}$　　转置后方阵$\boldsymbol{A}$

3. 矩阵相加。

提示：设有矩阵 $\boldsymbol{A}_{m\times n}$ 和矩阵 $\boldsymbol{B}_{m\times n}$，则其和亦为一个 m 行 n 列矩阵 $\boldsymbol{C}_{m\times n}$：

$$\boldsymbol{C}_{m\times n} = \boldsymbol{A}_{m\times n} + \boldsymbol{B}_{m\times n}$$

其中：

$$C_{ij} = A_{ij} + B_{ij} \quad (i = 1,2,\cdots,m; j = 1,2,\cdots,n)$$

可仿照本章中相应的例题自己设计算法，并用其编写程序用于计算 3 行 3 列的方阵之和。

4. 输入 10 个字符到一维字符数组 s 中，将字符串置逆。即 s[0]与 s[9]互换，s[1]与 s[8]互换……s[4]与 s[5]互换，输出置逆后的数组 s。

5. 替换加密(恺撒加密法)。加密规则是：将原来的字母用字母表中其后面的第 3 个字母的大写形式来替换，对于字母表中最后的 3 个字母，可将字母表看成是首末衔接的。例如，字母 c 就用 F 来替换，字母 y 用 B 来替换。请将字符串“I love you”译成密码。

6. 读入 5 个用户的姓名和电话号码，按姓名的字典顺序排列后，输出用户的姓名和电话号码。

7. 输入两个整型数组(假设数组的大小为 7)的各个元素，输出不是两个数组共有的元素。例如，输入 1 2 3 4 5 6 7 和 5 6 7 8 9 0，输出为 1 2 3 4 8 9 0。

8. 一个数组 A 中存有 $n(n>0)$ 个整数，在不允许使用另外的数组的前提下，将每个整数循环向右移 $m(m\geqslant0)$ 个位置，即将 A 中的数据由 $(A_0\ A_1\cdots A_{n-1})$ 变换为 $(A_{n-m}\cdots A_{n-1}\ A_0\ A_1\cdots A_{n-m-1})$(最后 m 个数循环移至最前面的 m 个数)。输入 $n(1\leqslant n\leqslant100)$、$m(m\geqslant0)$ 及 n 个整数，输出循环右移 m 位以后的整数序列。例如：

输入：

```
6 2
1 2 3 4 5 6
```

输出：

```
5 6 1 2 3 4
```

如果需要考虑程序移动数据的次数尽量少，要如何设计移动的方法?

提示：简单的思路是循环右移一位的操作重复进行 m 次即可，但这种做法的数据移动次数大约是 $m\times n$ 次。为了减少数据的移动次数，第二种方法是通过 3 次倒序来巧妙地实现。为简单起见，不妨设 $0\leqslant m<n$(否则先进行 $m\%=n$ 运算即可)，先把 $(A_0\ A_1\cdots A_{n-1})$ 倒序变成 $(A_{n-1}\ A_{n-2}\cdots\ A_1\ A_0)$，再把它的前 m 个元素 $(A_{n-1}\ A_{n-2}\cdots\ A_{n-m})$ 倒序成 $(A_{n-m}\cdots A_{n-1})$，然后把后 $n-m$ 个元素 $(A_{n-m-1}\ A_{n-m-2}\cdots\ A_1\ A_0)$ 倒序成 $(A_0\ A_1\cdots A_{n-m-1})$。这样，整个数组就成了 $(A_{n-m}\cdots\ A_{n-1}\ A_0\ A_1\cdots A_{n-m-1})$，这就是我们想要的结果。这种做法每个数据被移动了 2 次，所以总的数据移动次数是 $2n$ 次。

事实上，还可以有移动次数更少的算法，可以通过分析每个数据原位置与目标位置之间的下标关系，将每个数据一次性定位。根据题目的要求，可以发现：任何位于数组下标 i 位置的数据，其目的地址是下标为 $(i+m)\%n$ 的位置，或者说第 $(i-m+n)\%n$ 位置的数据将移到第 i 个位置。由于所有数据都需要移动，因此数据之间形成了一个

移动环。在这个移动环内实现循环移动,可以将第一个数据放到临时变量 t 中,然后将第二个数据放到第一个数据的位置,第三个数据放到第二个数据的位置……最后将 t 放到最后一个数据的位置。同时,也可以发现,对于任意的正数 n 和 m(不妨设 $m<n$),需要移动的环的个数就是 n 和 m 的最大公约数 gcd(n,m)。基于上述思路就可以将每个数据一次性定位。

9. 定义一个名为 Circle 的结构体(圆),其数据成员是圆的外接矩形的左上角和右下角两点的坐标,计算该圆的面积。

10. 编写字符串查找函数 mystrchr,该函数的功能为在字符串(参数 string)中查找指定字符(参数 c),如果找到了则返回该字符在字符串中的位置,否则返回 0。然后编写主函数验证之。函数原型为

```
int mystrchr(char string[],int c);
```

11. 编写字符串反转函数 mystrrev,该函数的功能为将指定字符串中的字符顺序颠倒排列。然后编写主函数验证。
提示:求字符串长度可以直接调用库函数 strlen,但在程序首部应加上

```
#include<cstring>
```

函数原型为

```
void mystrrev(char string[])
```

该函数无须返回值。

12. 编写一组求数组中最大最小元素的函数。该组函数的原型为

```
int imax(int array[],int count);          //求整型数组的最大元素
int imin(int array[],int count);          //求整型数组的最小元素
```

其中参数 count 为数组中的元素个数,函数的返回值即求得的最大或最小元素之值。要求同时编写出主函数进行验证。

13. 编写一组函数来实现词频统计功能:输入一系列英文单词,单词之间用空格隔开,用"xyz"表示结束输入,统计输入过哪些单词以及各单词出现的次数,统计时区分大小写字母,最后按单词的字典顺序输出单词和出现次数的对照表。(提示:利用结构体来描述单词和词频。)

14. 编写函数 isprime(int a)用来判断变量 a 是否为素数,若是素数,函数返回 1,否则返回 0。调用该函数找出任意给定的 n 个整数中的素数。

15. 编写一个猜数字的程序。程序选择 1～1000 之间的一个随机数,让玩家猜。它显示提示"Guess a number between 1 and 1000",玩家输入猜测的数字,如果错误,则提示猜得太大了(Too high)或太小了(Too low),帮助玩家继续猜测。如果猜对显示"Congratulation",并允许玩家选择是否再玩一次。
提示:使用函数 rand,该函数会返回一个随机数值,范围在 0 至 RAND_MAX 间。RAND_MAX 定义在 stdlib.h,其值为 2 147 483 647。但是 rand 函数生成的随机数严格意义上来讲只是伪随机数(pseudo-random integral number)。生成随机数时需

要指定一个种子(seed),如果在程序内循环,那么下一次生成随机数时调用上一次的结果作为种子。但如果分两次执行程序,那么由于种子相同,生成的"随机数"也是相同的。所以在调用 rand 函数产生随机数前,必须先利用 srand 设好随机数种子,如果未设随机数种子,rand 在调用时会自动设随机数种子为 1。所以,如果在调用 rand 之前没有调用 srand,则每次随机数种子都自动设成相同值 1,进而导致 rand 所产生的随机数值都一样。在实际应用中,一般取当前的时间作为第一个随机数的种子。使用函数 time(0)(在 time.h 中定义)得到当前时间。示例代码如下(产生 10 个 0～99 之间的随机数):

```
srand((unsigned int)time(0));
for(int i=0;i<10;i++)
    cout<<rand()%100<<endl;
```

16. Craps 游戏模拟。游戏规则如下。两人通过掷两枚骰子决定输赢,一方为庄家,另一方为玩家。在一局中,始终由玩家掷骰子。每个骰子有 6 个面,包含 1～6 点。等两枚骰子停止转动后,计算两个朝上面的点数之和。如果第一次掷出的点数和为 7 或者 11,则玩家胜。如果点数为 2、3 或者 12(称 Craps),庄胜。如果第一次掷出的点数和为 4、5、6、8、9 或 10,则该点数为玩家所需要的"正点"。玩家继续掷骰子,直到再次出现该"正点",则玩家胜。如果在玩家再次掷出"正点"之前出现了点数 7,则庄家胜。
17. 计算机在教育领域的使用被称为计算机辅助教学(CAI)。编写程序,帮助小学生练习 100 以内的正整数加法。程序产生 2 个 100 以内的正整数 a 和 b,在屏幕显示"a+b=?",然后,学生输入答案。如果答对,显示"Very Good!",并出下一道题;如果错误,显示"NO",然后让学生重新给出答案,直到做对为止。要求使用一个独立的函数产生每一道题。
18. 在上一题的基础上,要求 $a+b\leqslant 100$,如果可以出 72+8 的题,但不能出 56+64 的题。当学生连续做对 20 题后程序自动结束。
19. 编写程序,将某一个输入的位数不确定的正整数按照标准的三位分节格式输出。例如,当用户输入 82668634 时,程序应该输出 82,668,634。
20. 编写程序,把 10 个整数 1、2、…、10 赋予某个 int 型数组,然后用 int 型指针输出该数组元素的值。
21. 用指针编写一个程序,当输入一个字符串后,要求不仅能够统计其中字符的个数,还能分别指出其中大小写字母、数字以及其他字符的个数。
22. 编写一个函数,用于将一个字符串转换为整型数值。其原型为

```
int atoi(char * string);
```

其中参数 string 为待转换的字符串(其中包括正、负号和数字),返回值为转换结果。

23. 编写一个函数,用于生成一个空白字符串,其原型为

```
char * mystrspc(char * string,int n);
```

其中参数 string 为字符串,n 为空白字符串的长度(空格符的个数)。返回值为指向 string 的指针。

24. 设计一个函数 char * f(char * p,char * s),功能是在 p 指向的字符串中查找并删除 s 指向的子串。若找到多个子串则必须全部删除,删除完成后返回结果字符串的首地址。函数 main 输入字符串与子串,调用 f 后输出结果字符串。

25. 输入若干有关颜色的英文单词,以 # 作为输入结束标志,其中单词数小于 20,每个单词不超过 10 个字母。要求对这些单词按字典顺序排序后输出。

26. Ackermann 函数 ack(m,n)采用以下递归形式定义:

ack(0,n)=$n+1$

ack(m,0)=ack($m-1$,1)

ack(m,n)=ack($m-1$,ack(m,$n-1$))

其中,$m>0$,$n>0$。

编写一个计算此函数的递归程序。

第 7 章 算法分析与设计

引言

算法可以理解为由基本运算及规定的运算顺序所构成的完整的解题步骤，或者看成按照要求设计好的有限的、确切的计算序列，并且这样的步骤和序列可以解决一类问题。算法是解题方案的准确而完整的描述，是一系列解决问题的清晰指令。算法代表着用系统的方法描述解决问题的策略机制，也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。如果一个算法有缺陷，或不适合某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂度与时间复杂度来衡量。

教学目标

- 掌握常用的查找算法。
- 掌握常用的排序算法。
- 掌握常用查找和排序算法的时间复杂度分析。
- 对常用算法有一定的了解。

7.1 算法的基本概念

算法（algorithm）一词来源于阿拉伯数学家 AIKhowarizmi 编写的《波斯教科书》（*Persian Textbook*），书中概括了进行算术四则运算的法则。后来的《韦氏新世界词典》将其定义为“求解某种问题的任何专门的方法”。

对复杂的系统性问题，算法设计是在通过需求分析建立起相应的业务模型和数学模型，以及通过模块设计确定了相应的系统结构和数据结构的基础上，对模块功能的进一步细化，是求解问题的方法的描述。算法设计包括确定算法的控制结构（顺序、循环或选择）以及实现的具体步骤和操作。算法设计的正确与否和精练与否决定了程序编码的正确性和有效性。

利用算法求解问题的一般思路如图 7-1 所示。

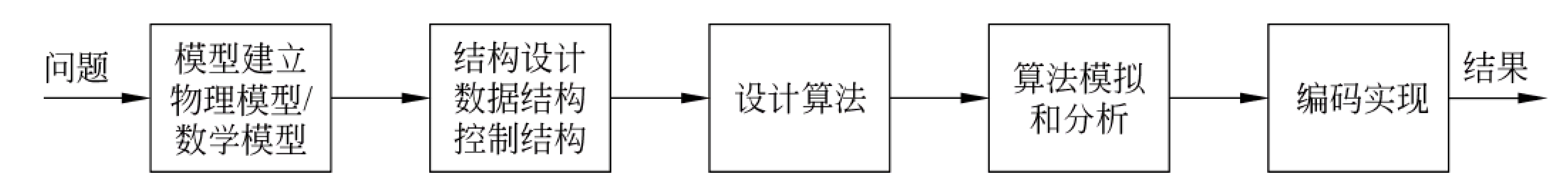


图 7-1　利用算法求解问题的一般思路

算法是一个有穷规则的集合，它表现为一个解决某个特定类型问题的运算序列。通俗地讲，算法定义了解决某个问题的一系列步骤或方法，如果遵循它就可以完成一项特定的任务。算法是定义在逻辑结构上的操作，是独立于计算机的，而它的具体实现则是在计算机上进行的。

通过前面的学习，可以总结出算法应具有如下特性：

(1) 有穷性。一个算法必须在执行有穷步后结束，且每一步都能在有限的时间内完成。即一个算法所包含的计算步骤和时间都是有限的。

(2) 确定性。算法的每一个步骤都必须具有确切的定义，即算法中所有有待执行的动作都必须有严格的、毫不含混的规定，不能有歧义。

(3) 能行性(或称可行性)。算法中所有有待实现的运算都必须是能够精确执行的，且用纸和笔做有穷次即可完成。算法的执行者甚至不需要掌握算法的含义即可根据算法的每个步骤要求进行操作，并最终得出正确的结果。

(4) 输入。一个算法应该有 0 个或多个输入。

(5) 输出。一个算法应该有 1 个或多个输出。

【例 7-1】 设计求 2＋4＋6＋…＋10000 的算法。

算法描述如下：

步骤 1：让变量 sum＝0。

步骤 2：让变量 j＝2。

步骤 3：计算 sum＋j，结果仍放在 sum 中，即让 sum＝sum＋j。

步骤 4：让 j＝j＋2。

步骤 5：如果 j 不大于 10000，返回执行步骤 3，否则执行下一步。

步骤 6：输出结果 sum 的值。

在例 7-1 中，步骤 3 至步骤 5 重复执行了 4999 次，这就是循环结构。另外，步骤 5 是一个逻辑判断，判断的结果导致两种可能的执行流程，一种是向上循环执行，另一种是向下执行，这就是选择结构。

对于求 2＋4＋6＋…＋10000 的算法，还可以有其他计算方法求解。比如利用公式来计算，即只要计算(1＋5000)×5000。这样一来，算法就只有 3 步：先计算加法，再计算乘法，最后输出结果。

可见，算法设计是非常灵活的，对同一个问题可以有不同的算法描述。但不同的算法可能有不同的效率。对于复杂问题，算法就更重要了。要在保证求解问题正确的前提下，尽可能地追求算法的效率，也就是要尽可能地设计出复杂度低的算法。

7.2　算法的描述方法

算法的表示方法有多种，最简单的就是自然语言表示法。除此之外，常用的描述方法还有伪代码、流程图等。

7.2.1 算法的自然语言描述

所谓自然语言就是人们在日常生活中使用的语言，比如汉语、英语、日语和俄语等。对初学者来说，用自然语言描述算法最为直接，没有语法和语义障碍，容易理解。但用自然语言描述算法文字冗长，不够简明，尤其会出现含义不太严格的情况，要根据上下文才能判断出正确的含义。

【例 7-2】 描述求任意两个正整数的最大公因数的算法。

先来看一下著名的欧几里得算法。古希腊数学家欧几里得曾给出了求解两个数的最大公因子的算法描述：

步骤 1：如果 $p<q$，交换 p 和 q。

步骤 2：求出 p/q 的余数 r。

步骤 3：如果 $r=0$，则 q 就是所求的结果；否则反复做如下工作：

令 $p=q$，$q=r$，重新计算 p 和 q 的余数 r，直到 $r=0$ 为止。q 就是原来的两个正整数的最大公因数。

下面将欧几里得算法进一步细化为以下的步骤：

步骤 1：输入两个正整数，分别放在变量 p 和 q 中。

步骤 2：如果 $p<q$，则交换 p 和 q 的值(即让 $r=p$，$p=q$，$q=r$)。

步骤 3：将 p/q 的余数放在 r 中(即让 $r=p/q$ 的余数)。

步骤 4：如果 r 等于 0，则执行步骤 6，否则执行步骤 5。

步骤 5：让 $p=q$，$q=r$，执行步骤 3。

步骤 6：输出 q 的值。

7.2.2 算法的伪代码描述

伪代码(pseudo code)介于自然语言和计算机语言之间，用编程者熟悉的计算机语言的语句加上自然语言构成(尽可能地融入编程语言的函数和语法)，基本上可以随心所欲地写。例如，输入并比较两个学生成绩的过程可用类 C 语言的语法描述如下(对一个班的成绩处理要更复杂一些)：

```
cout <<请输入学生姓名、学号、英语成绩、数学成绩
cin >>姓名 1 >>学号 1 >>英语成绩 1 >>数学成绩 1
合计 1=英语成绩 1+数学成绩 1
cout <<请输入学生姓名、学号、英语成绩、数学成绩
cin >>姓名 2 >>学号 2 >>英语成绩 2 >>数学成绩 2
合计 2=英语成绩 2+数学成绩 2
if(合计 1>合计 2)
    if(英语成绩 1>0 并且 数学成绩 1>0)
        cout <<姓名 1 <<学号 1
else
```

```
if(英语成绩 2>0 并且 数学成绩 2>0)
    cout <<姓名 2 <<学号 2
```

思考 这个算法是有缺陷的,你看出来了吗?

此例子相当简单,接触过 C 语言的人可以看出,这基本上就接近程序本身了。

7.2.3 算法的流程图描述

所谓流程图(flow chat),是用几种几何图形、线条和文字来表示不同的操作和处理步骤。用流程图表示算法形象直观,简洁清晰,易于理解。美国国家标准化协会(American National Standard Institute,ANSI)规定了常用流程图符号,如图 7-2 所示。

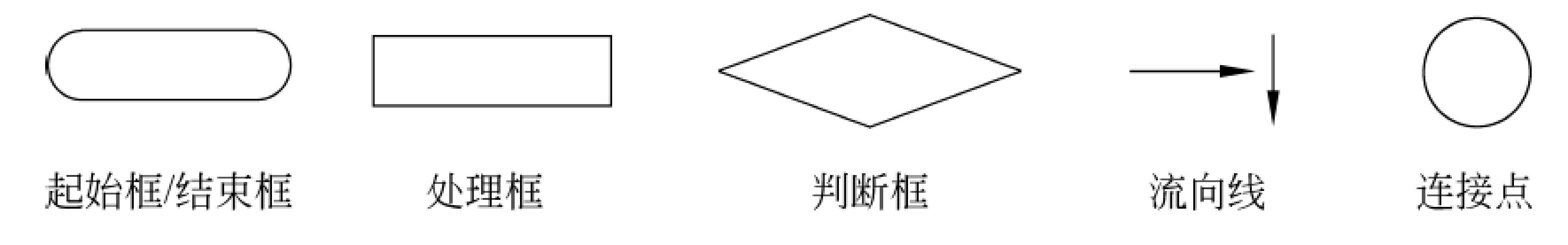


图 7-2 常用流程图符号

上述"输入并比较两个学生成绩"的算法用流程图方式描述如图 7-3 所示。由图中可以比较清楚地看出该算法描述存在的问题,即存在没有输出信息的可能,而算法的基本特性之一是要求至少有一个输出。

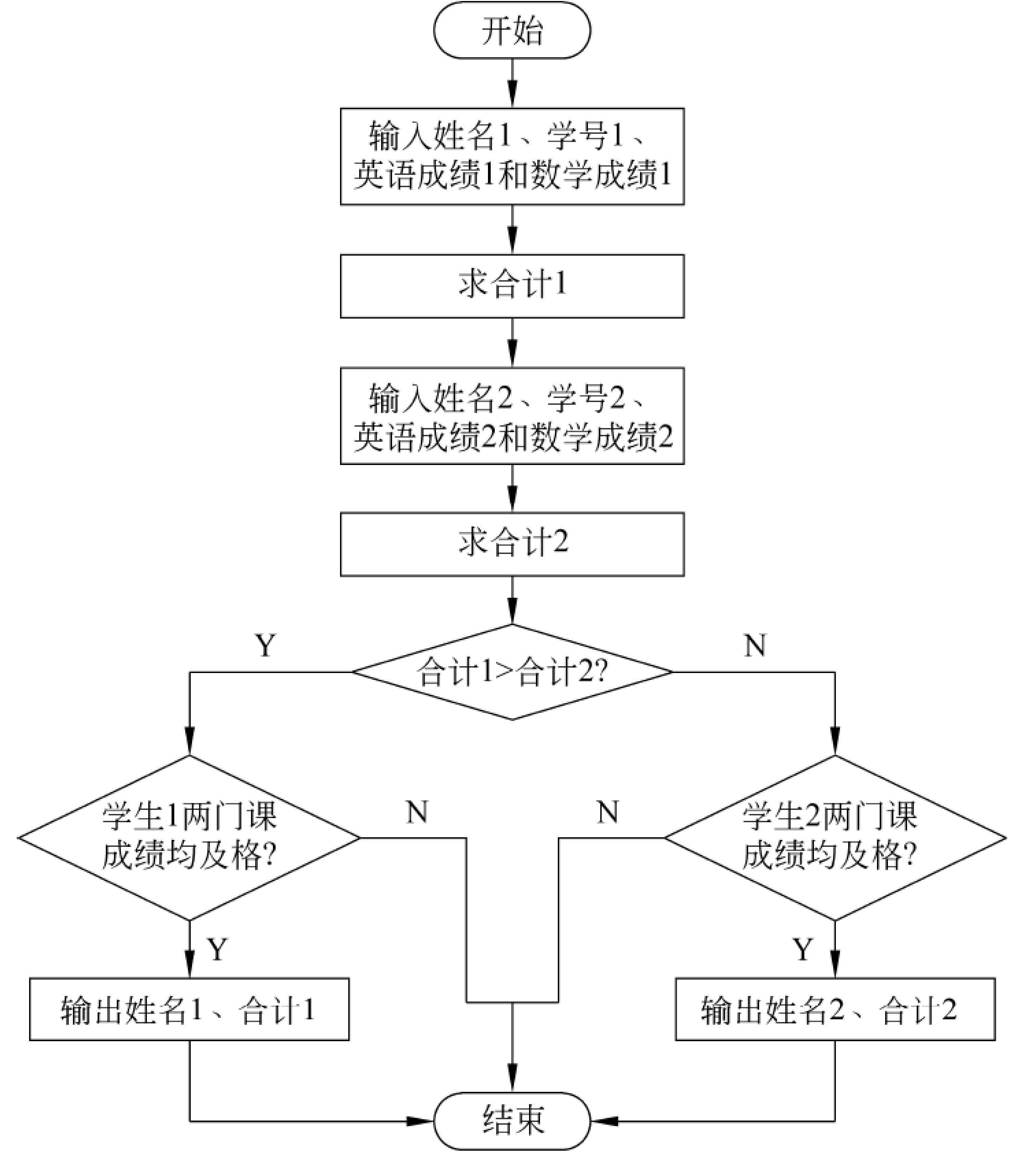


图 7-3 输入并比较两个学生的算法流程

思考 请考虑如何修改。

【**例 7-3**】 用流程图描述欧几里得算法。

解：欧几里得算法流程如图 7-4 所示。需要注意的是图中的“R＝P，P＝Q，Q＝R”处理框也可以分解成 3 个处理框。

比较用流程图表示的欧几里得算法与例 7-2 中用自然语言描述的算法，不难发现，流程图描述的算法逻辑清晰，直观形象，易于理解。

关于流程的详尽程度，并没有一个绝对统一的标准，因此算法设计的结果并不唯一。对于初学者来说，只要能正确求解问题就可以。

在画流程图（即设计算法）时，往往会出现一张纸由上而下画满了，但算法描述还未结束的情况，这时候就要将连接点符号画在纸张的底部，然后在另一张白纸的头部也画同样的连接点符号，这就意味着两张算法流程图被拼接起来，形成一幅完整的流程图。当然也会出现纸张左右画满的情况，这时候也需要用连接点符号。判断框有一个入口和两个出口，两个出口的条件总是截然相反的，一个若代表条件成立，则另一个代表条件不成立。只要在两个出口流向线之一的旁边标注清楚即可。

下面再来看一个利用流程图描述算法的示例。

【**例 7-4**】 用流程图描述求解 1 － 1/2＋1/3 － 1/4＋1/5 － 1/6＋…＋1/99 － 1/100 的算法。

描述本例算法的流程图如图 7-5 所示。

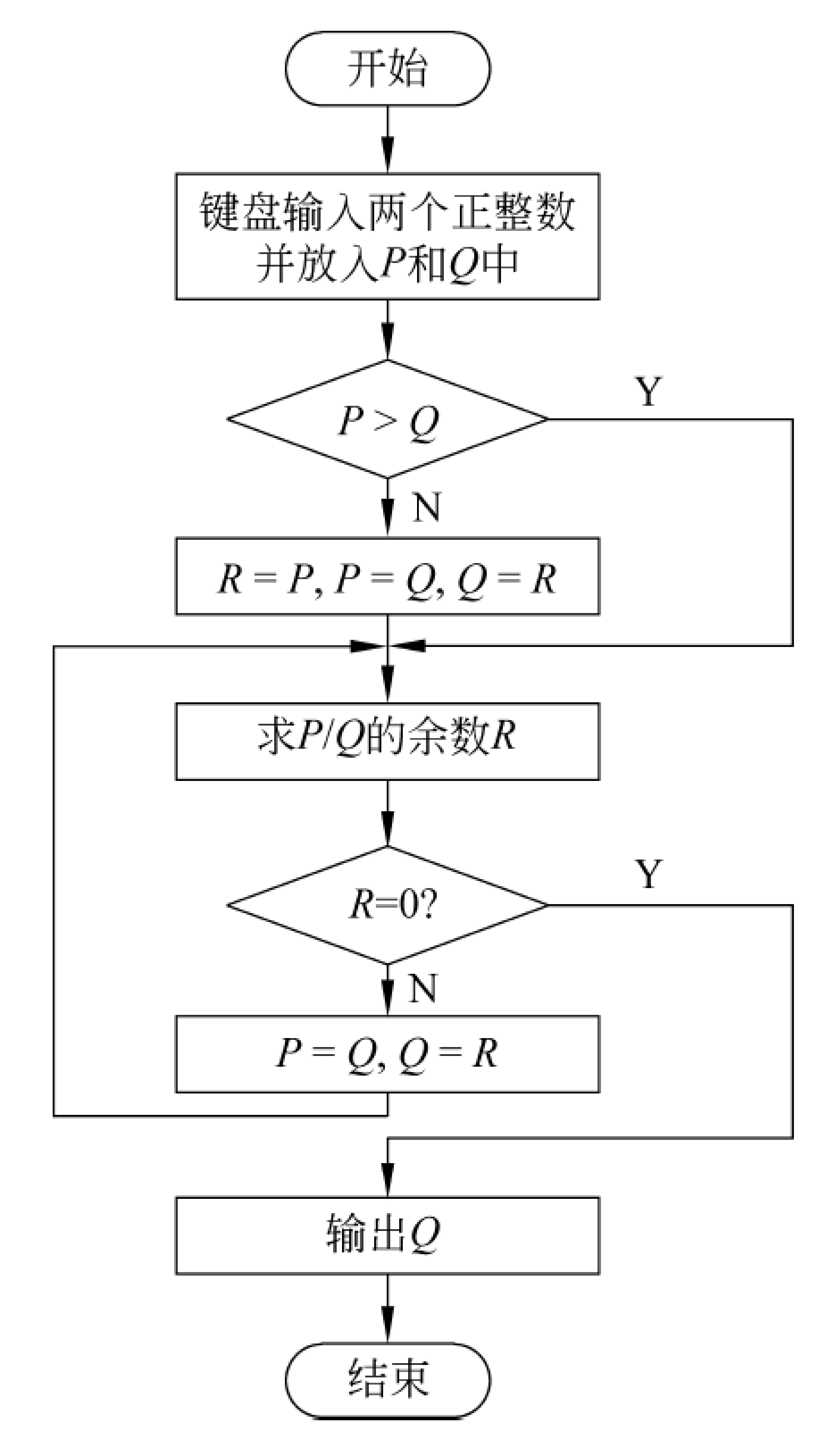


图 7-4 欧几里得算法流程图

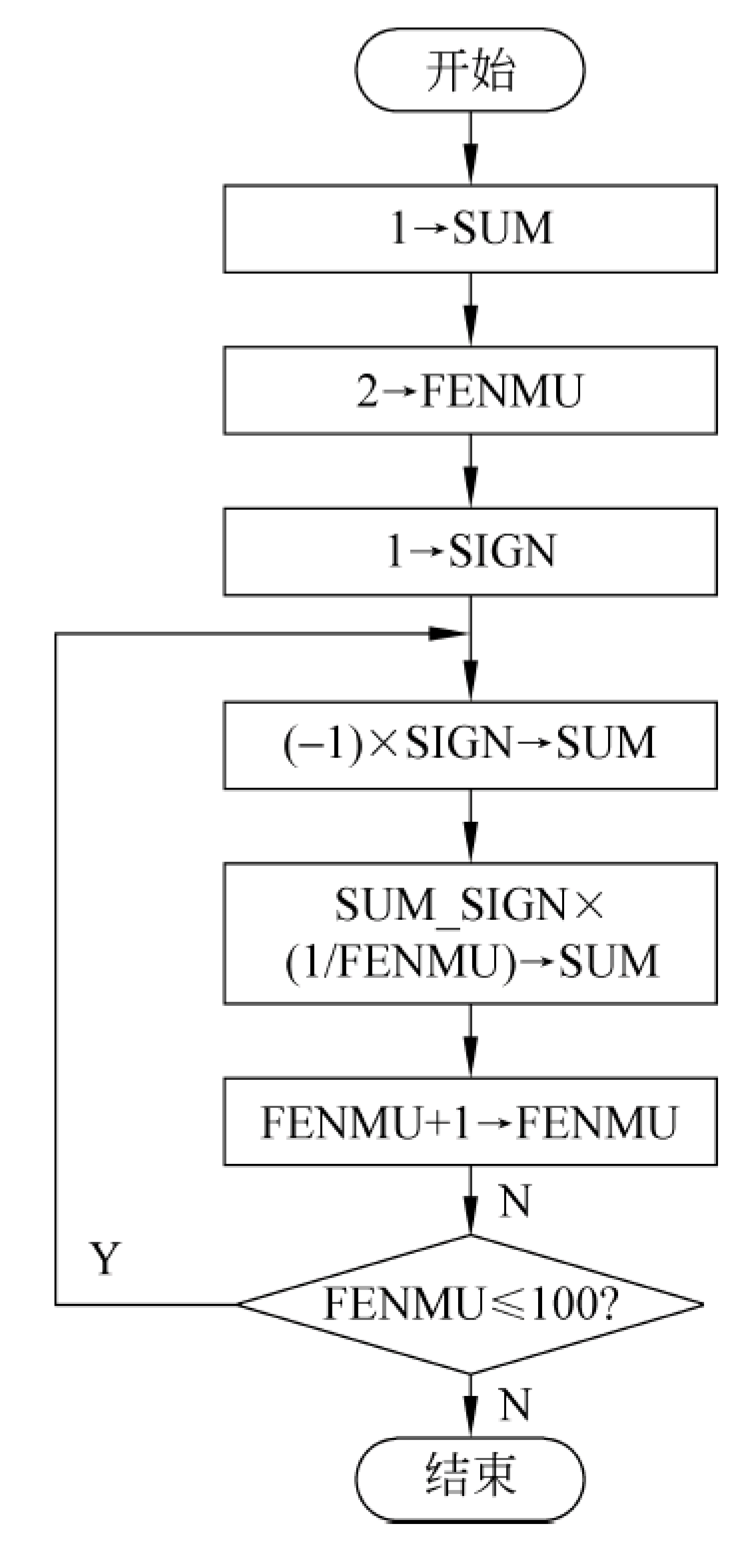


图 7-5 例 7-4 算法流程图

7.3 算法的复杂性评价

解决同样一个问题可以用不同的算法。一个算法的质量优劣将影响到算法乃至程序的效率。

对算法的分析和评价一般应考虑正确性、可维护性、可读性、运算量及占用存储空间等诸多因素。通常,在算法正确性的前提下,评价一个算法的主要指标如下:

(1) 算法实现所消耗的时间,即时间复杂度。

(2) 算法实现所消耗的存储空间,即空间复杂度。

(3) 算法应易于理解、易于编码、易于调试等。

7.3.1 算法的时间复杂度

1. 时间频度

要确定一个算法执行所耗费的时间,最直接的方法就是测试。但是不可能也没有必要对每个算法都上机测试,只需知道哪个算法花费的时间多,哪个算法花费的时间少就可以了。

假定可以知道算法中每一条语句执行一次所需的平均时间,则有

算法运行所需的时间 = 语句执行一次所需的平均时间 × 语句执行次数　(7.1)

语句执行一次所需的平均时间取决于计算机 CPU 的主频、是否分时系统、编译系统的效率和优化程度、输入/输出速度等不确定因素。而一般来说,一个算法中语句的执行次数是确定的。由式(7.1)可知,一个算法花费的时间与算法中语句的执行次数成正比,即算法中语句执行次数越多,它花费的时间就越多。一个算法中的语句执行次数称为语句频度或时间频度,记为 $T(n)$。

2. 时间复杂度

时间频度中的 n 称为问题的规模(大小),比如 n 条语句指令、n 个子程序、n 个功能模块等所需的执行时间。当 n 不断变化时,时间频度 $T(n)$ 也会不断变化。但有时我们想知道它变化时呈现什么规律,为此引入时间复杂度的概念。

一般情况下,算法中基本操作重复执行的次数是问题规模 n 的某个函数,用 $T(n)$ 表示,若有某个辅助函数 $f(n)$,使得当 n 趋近于无穷大时,有

$$\lim_{n \to x} \frac{T(n)}{F(n)} = M \quad (M\text{ 为正的常数})$$

则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n)=O(f(n))$,称 $O(f(n))$ 为算法的渐进时间复杂度,简称时间复杂度。

按数量级增序排列,常见的几种时间复杂度有常数阶 $O(1)$、线性阶 $O(n)$、对数阶 $O(\log n)$、线性对数阶 $O(n \log n)$、平方阶 $O(n^2)$、立方阶 $O(n^3)$、k 次方阶 $O(n^k)$、指数阶

$O(2^n)$等。随着问题规模 n 的不断增大,上述时间复杂度也就不断增大,算法的执行效率就不断降低。

在各种不同算法中,若算法中语句执行次数为一个常数,则时间复杂度为 $O(1)$。而下面的 C 语句可认为是 $O(n^2)$时间复杂度:

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        A(i,j)=0
```

值得指出的是:在时间频度不相同时,时间复杂度有可能相同。例如,$T(n)=n^2+3n+4$ 与 $T(n)=4n^2+2n+1$,它们的频度不同,但时间复杂度相同,都为 $O(n^2)$。

不同的复杂度函数之间的对比如表 7-1 所示。

表 7-1 不同的复杂度函数值对比

n	$\log n$	$n \log n$	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65 535
32	5	160	1024	4 294 967 296

7.3.2 算法的空间复杂度

算法的空间复杂度是算法在计算机内执行时所需存储空间的度量。一个算法的实现所占用的存储空间大致有 3 个方面:一是指令、常数、变量所占用的存储空间;二是输入数据所占用的存储空间;三是算法执行时必需的辅助空间。前两个空间是计算机运行时所必需的。因此,把算法在执行时所需的辅助空间的大小作为分析算法空间复杂度的依据。

与算法时间复杂度的表示一致,将算法中所用辅助空间大小的数量级来表示算法的空间复杂度,仍然记为 $O(n)$。常见的几种空间复杂度有 $O(\log n)$,$O(n)$,$O(n^2)$,$O(2^n)$等。

事实上,对一个问题的算法实现,时间复杂度和空间复杂度往往是相互矛盾的,要降低算法的执行时间就要以使用更多的空间为代价,要节省空间就可能要以增加算法的执行时间作代价,两者很难兼顾。因此,只能根据具体情况有所侧重。

7.4 查找算法

“数据查找”是经常碰到的问题之一。所谓查找,就是根据给定的关键字值(就是数据元素中可以唯一标识一个数据元素的数据项,如学生的学号、居民身份证号码等),在一组

数据中确定一个其关键字值等于给定值的数据元素。若存在这样的数据元素，则称查找是成功的；否则称查找不成功。一组待查数据元素的集合又称为查找表。在查找过程中，查找表一旦建立不再改变的查找称为静态查找，反之则是动态查找。在此仅讨论静态查找。

7.4.1 顺序查找

顺序查找是最普通也是最简单的查找技术。其基本思想是：从数组中的第一个元素开始，逐个把元素的关键字值和给定值比较，若某个元素的关键字值和给定值相等，则查找成功；否则，若直至第 n 个记录都不相等，说明不存在满足条件的数据元素，查找失败。

顺序查找算法的伪代码描述如下：

```
从第一个元素开始
while()
{
    比较当前元素的关键字值与所需关键字值
    if(找到了)
        返回当前元素
      if(所有元素找过了)
          返回查找失败
      下一个元素
}
```

在该算法中，执行频率最高的是 while 语句。当查找表中元素个数 n 很大时，其平均查找长度 $ASL=(n+1)/2$，即每次查找平均要比较一半数据元素。

【例 7-5】 在整数数组内顺序查找。

分析：从头开始，逐一匹配，找到为止。

程序：

```
# include< stdio.h>
int SqSearch(int * a, int n, int k);
int main()
{
    int a[]= {13, 67, 89, 2, 15, 99, 77, 56, 34};
    int k= SqSearch(a,9,15);                //搜索 15
    if(k> = 0)
    {
        printf("查找的数据在数组中的位置为: %d\n", k);
    }
    else
    {
        printf("数据没有找到\n");
    }
}
```

```
/* SqSearch函数在元素个数为n的数组中搜索特定的数,如果找到,则返回数据在数组中的位置,
否则返回-1 */
int SqSearch(int* a, int n, int k)
{
    for(int i=0;i<n;i++)
    {
        if(a[i]==k)
            return i;
    }
    return -1;
}
```

顺序查找的时间复杂度是 $O(n)$。算法的最坏情形是要检查每个元素,以判断数组中是否存在要查找的元素。如果数组长度加倍,则算法要执行的比较次数也会加倍。

7.4.2 折半查找

如果查找表中的所有数据元素都按关键字有序组织,则可以采用一种更高效的查找方法——折半查找(或称二分查找)。

折半查找的基本思想是:由于查找表中的数据元素按关键字有序(假设递增有序),则在查找时不必逐个顺序比较,而采用跳跃的方式——先与“中间位置”的记录关键字值比较,若相等,则查找成功;若给定值大于“中间位置”的关键字值,则在后半部继续进行折半查找;否则在前半部进行折半查找。

折半查找的过程是:先确定待查元素所在区域,然后逐步缩小区域,直到查找成功或失败为止。

设待查元素所在区域的下界为 low,上界为 high,则中间位置 mid=(low+high)/2。折半查找算法的描述如下:

步骤 1:设置查找的区间,令 low=0,high=n−1。

步骤 2:计算中间位置,mid=(low+high)/2。

步骤 3:若 key=A(mid),查找成功,返回 mid;若 key<A(mid),则令 high=mid−1 后执行步骤 2;若 key>A(mid),则令 low=mid+1 后执行步骤 2。

步骤 4:若当 low=high 时,key 不等于 A(mid),则查找失败,返回−1。

由于折半查找要求数据元素的组织方式应具有随机存取的特性,所以折半查找只适用于以顺序结构组织的有序查找表。折半查找成功的平均查找长度 ASL≈log(n+1)−1。

折半查找的优点是比较次数少,查找速度快。但为了快速查找所付出的代价是要对数据元素按关键字值的大小进行排序,而排序一般是很费时的,所以折半查找适用于一经建立就很少变动而又经常要进行查找的有序表。

【例 7-6】 在有序数组 A 中折半查找。

程序:

```
# include< stdio.h>
int BinSearch(int * a,int n,int key);
int main()
{
    int a[]= {3,5,11,22,34,56,76,87,90,92,95,123,134};
    int k= BinSearch(a,13,95);                    //搜索 95
    if(k> 0)
        printf("查找的数据在数组中的位置为:% d\n",k);
    else
        printf("数据没有找到\n");
    return 0;
}
/* BinSearch 函数在数组中搜索特定的数,如果找到,则返回数据在数组中的位置,否则返回 * /
int BinSearch(int * a,int n,int key)
{
    int low= 0;
    int high= n- 1;
    while(low< = high)
    {
        int mid= (low+ high)/2;
        if(key = a[mid])
            return mid                            //找到
        else
            if(key < A[mid])
                high = mid - 1                    //在前半部分继续寻找
            else
                low = mid+ 1                      //在后半部分继续寻找
    }
    return - 1                                    //没有找到
}
```

7.5 排 序 算 法

排序是计算机内经常进行的一种操作,其目的是将一组无序的记录序列调整为有序的记录序列。排序可以分为内部排序和外部排序。若整个排序过程不需要访问外存便能完成,则称此类排序问题为内部排序。反之,若参加排序的记录数量很大,整个序列的排序过程不可能在内存中完成,则称此类排序问题为外部排序。内部排序的过程是一个逐步扩大记录的有序序列长度的过程。本节仅讨论内部排序,同时假设要排序的数据均存储在一个一维数组内。

7.5.1　冒泡排序

冒泡排序的基本思想是两两比较待排序记录的关键字，发现两个记录的次序相反时即进行交换，直到没有反序的记录为止。

设想将被排序的数组 R 垂直排列，数组中每个元素 $R[i]$的值看作是重量为该值的气泡。根据轻气泡不能在重气泡之下的原则，从下往上扫描数组 R。凡扫描到违反本原则的轻气泡，就使其向上“飘浮”（也就是交换轻气泡和重气泡的位置）。如此反复进行，直到最后任何两个气泡都是轻者在上、重者在下为止。此时数组排序完毕。具体步骤如下：

步骤 1：假设带排序的数存于数组 R 中（R 的下标范围为 0 到 n）。

步骤 2：第一趟扫描：从 R 的结尾处开始，依次比较相邻的两个数值的大小，若发现小者在下、大者在上，则交换二者的位置。即依次比较（$R[n]$，$R[n-1]$），（$R[n-1]$，$R[n-2]$），…，（$R[1]$，$R[0]$），对于每对气泡（$R[j+1]$，$R[j]$），若 $R[j+1]<R[j]$，则交换 $R[j+1]$和 $R[j]$的内容。

步骤 3：当第一趟扫描完毕时，最小的数值就飘浮到该数组的顶部，即最小的数组元素被放在位置 $R[0]$上。

步骤 4：第二趟扫描：类似于第一趟扫描，只不过扫描的范围从 $R[1]$到 $R[n]$，扫描的结果将使次小的数存放于 $R[1]$中。

步骤 5：经过 n 趟扫描，可以得到排序后的数组 R。

假设数组 R 具有 5 个整数元素，分别是 34、12、2、77 和 68，如图 7-6 所示。

R[0]	R[1]	R[2]	R[3]	R[4]
34	12	2	77	68

图 7-6　待排序的数组

第一趟冒泡排序扫描的过程如图 7-7 所示。其中，(a)中 68<77，因此 68 和 77 交换位置；(b)中 68>2，因此不发生交换；(c)和(d)中 2 被交换到了最前端的位置上。经过第一趟扫描后，2 将上浮到 $R[0]$位置。

R[0]	R[1]	R[2]	R[3]	R[4]
34	12	2	68	77

(a)

R[0]	R[1]	R[2]	R[3]	R[4]
34	12	2	68	77

(b)

R[0]	R[1]	R[2]	R[3]	R[4]
34	2	12	68	77

(c)

R[0]	R[1]	R[2]	R[3]	R[4]
2	34	12	68	77

(d)

图 7-7　第一趟冒泡排序扫描过程

随后再经过同样的 3 趟扫描，12、34 和 68 将被交换到正确的位置上，排序就完成了。

【例 7-7】　编写程序，用于整数数组的冒泡排序。

分析：设有大小为 M 的整数数组，编写一个过程，用冒泡排序法对该数组排序，过程参看前面描述的步骤。

程序：

```
# include< stdio.h>
void Show(int * s,int n);
void Bubble(int * s,int n);
int main()
{
    int s[]= {12,- 78,67,23,2,99,234,- 23,45,56,12,78};
    //在屏幕上显示数组
    Show(s,12);
    //排序并显示排序后的结果
    printf("数组排序 ...\n");
    Bubble(s,12);
    //显示排序后的结果
    Show(s,12);
    return 0;
}
//在屏幕上显示数组
void Show(int * s,int n)
{
    for(int k= 0;k< n;k+ + )
        printf("% d", * (s+ k));
    printf("\n");
}
//冒泡排序算法
void Bubble(int * s,int n)
{
    for(int i= 0;i< n;i+ + )
        for(int j =n- 1; j> i; j- - )
            if(s[j] < s[j- 1])
            {    //交换
                int temp= s[j];
                s[j]= s[j- 1];
                s[j- 1]= temp
            }
}
```

从上面的代码可以看到，通过两重循环比较元素之间的大小，若需要排序的元素个数为 n，则比较的次数为

$$(n-1)+(n-2)+\cdots+1=n(n-1)/2=(n^2-n)/2$$

最差的情况，每次比较都需要交换，其时间复杂度为 $O(n^2)$。在例 7-7 的冒泡排序上可以有一个小的改进，即当某趟比较的时候如果没有交换发生，说明排序已经完成(见本章习题的第 2 题)。

7.5.2 选择排序

选择排序算法的思想是：第一次从数组中选择最小的元素，并将它与第一个元素交换；第二次选择剩余元素中最小的（是所有元素中第二小的），并将其与第二个元素交换……一直这样做下去，直到最后一次选择了第二大的元素，并将它与倒数第二个位置的元素交换（如果有必要），使最大的元素位于最后一个位置。经过 i 次选择和交换后，数组中前 i 个元素将按照升序保存在数组的前 i 个位置中。

假设数组 R 具有 5 个整数元素，分别是 34、12、2、77 和 68，如图 7-8 所示。

选择排序程序首先判断出最小的元素为 2 位于索引 2 处（即第 3 个元素最小），于是程序首先将 2 和 34 交换。接下来可以确定剩余的元素中最小的 12 处于正确的位置（不用交换）……该过程一直继续直到完成整个排序，如图 7-9 所示。

R[0]	*R*[1]	*R*[2]	*R*[3]	*R*[4]
34	12	2	77	68

图 7-8 待排序的数组

	R[0]	*R*[1]	*R*[2]	*R*[3]	*R*[4]
第1次	2	12	34	77	68
第2次	2	12	34	77	68
第3次	2	12	34	77	68
第4次	2	12	34	68	77

图 7-9 选择排序的过程

【例 7-8】 编写程序，用于整数数组的冒泡排序。

分析：设有大小为 M 的整数数组，编写一个过程，用选择排序法对该数组排序，过程参看前面描述的步骤。

程序：

```
#include<stdio.h>
void Show(int * s,int n);
void SelectionSort (int * s,int n);
int main()
{
    int s[]={12,-78,67,23,2,99,234,-23,45,56,12,78};
    //在屏幕上显示数组
    Show(s,12);
    //排序并显示排序后的结果
    printf("数组排序 ...\n");
    SelectionSort (s,12);
    //显示排序后的结果
    Show(s,12);
    return 0;
}
```

```
//在屏幕上显示数组
void Show(int* s,int n)
{
    for(int k=0;k<n;k++)
        printf("%d ",*(s+k));
    printf("\n");
}
//选择排序
void SelectionSort(int* s,int n)
{
    //最小一个数的索引
    int smallest;
    for(int i=0;i<n-1;i++)
    {
        Smallest=i;
        for(int j=i+1; j<n;j++)
        {
            if(s[j] <s[smallest])
                smallest=j;
        }
        //交换
        int temp=s[smallest];
        s[smallest]=s[i];
        s[i]=temp;
    }
}
```

选择排序的时间复杂度和冒泡排序的时间复杂度是一样的，为 $O(n^2)$。和冒泡排序一样，选择排序也经过了 $(n^2-n)/2$ 次比较。

7.5.3 快速排序

快速排序(quicksort)是对冒泡排序的一种改进，由 C. A. R. Hoare 在 1962 年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

快速排序的基本算法如下：设要排序的数组是 $A[0], A[1], \cdots, A[N-1]$，首先任意选取一个数据(通常选用第一个数据)作为关键数据，然后将所有比它小的数都放到它前面，所有比它大的数都放到它后面，这个过程称为一趟快速排序。

一趟快速排序的算法如下：

步骤 1：设置两个变量 i 和 j，排序开始的时候 $i=0, j=N-1$。

步骤 2：以第一个数组元素作为关键数据，赋值给 key，即 key$=A[0]$。

步骤 3：从 j 开始向前搜索，即由后开始向前搜索（$j=j-1$），找到第一个小于 key 的值 $A[j]$，令 $A[i]=A[j]$。

步骤 4：从 i 开始向后搜索，即由前开始向后搜索（$i=i+1$），找到第一个大于 key 的值 $A[i]$，令 $A[j]=A[i]$。

步骤 5：重复步骤 3～5，直到 $i=j$。

步骤 6：令 $A[i]=\text{key}$。

设有数组 $A=\{50,39,64,90,72,12,29\}$，一趟快速排序的交换过程如图 7-10 所示。

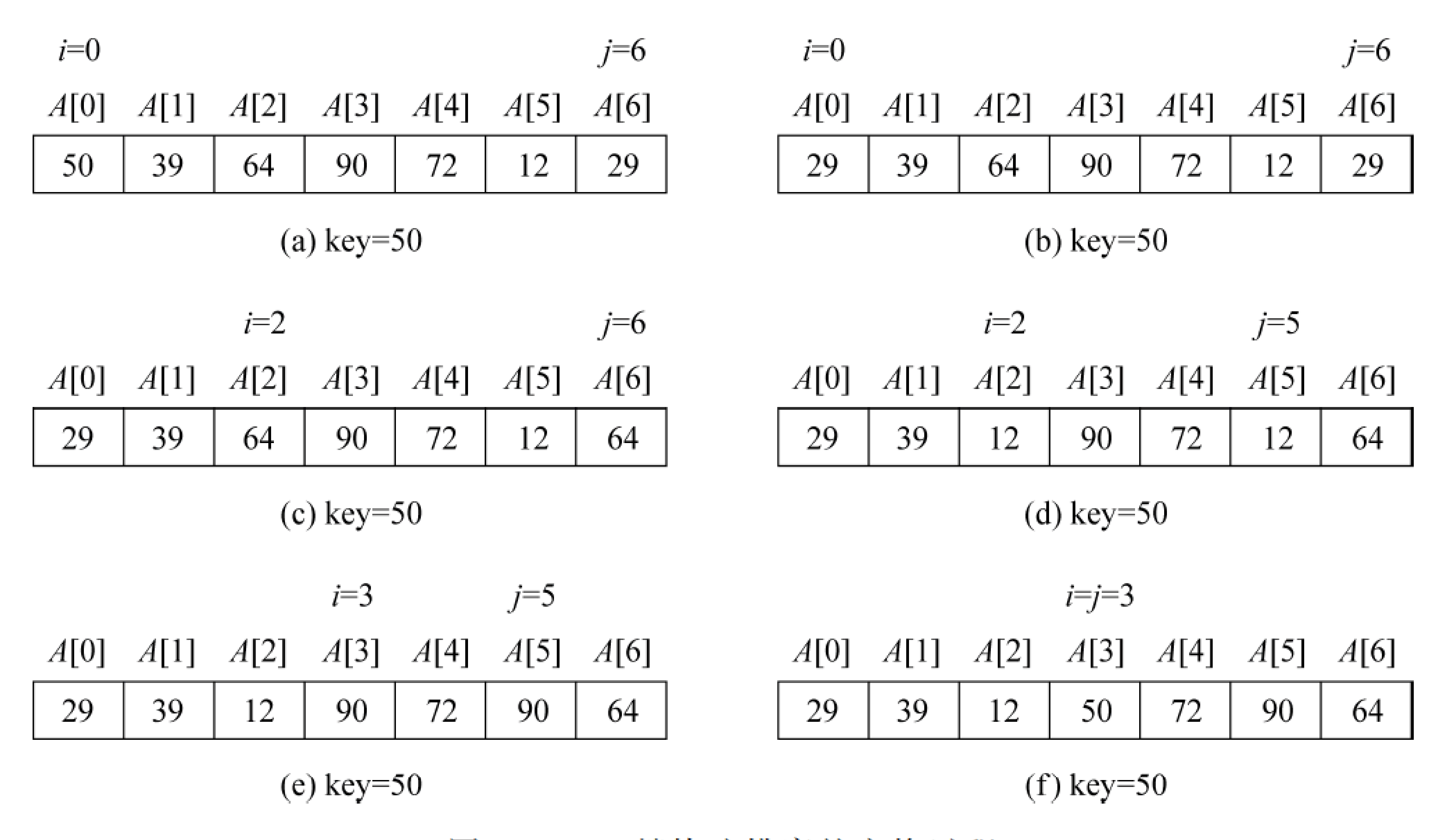


图 7-10　一趟快速排序的交换过程

在图 7-10 中，(a)是初始状态。在(b)中，由于 $A[6]<\text{key}$，因而被放置到 $A[0]$。在(c)中，当 $i=1$ 时，没有改变，直到 $i=2$，$A[2]$的值放置到 $A[6]$。(d)和(e)重复这一过程，直到(f)，此时 $i=j=3$，将 key 的值放入。经过这样的一趟排序后，凡是比 key=50 大的值都移动到数组的后半部分，比 50 小的都在前面。

要完成整个数组的排序，只需要递归调用此过程，以 50 为中点分割这个数据序列，分别对前一半和后一半进行类似的快速排序，从而完成全部数据序列的快速排序，也就是对 $A[0]$到 $A[2]$排序，对 $A[4]$到 $A[6]$排序，最后把此数据序列变成一个有序的序列。

【例 7-9】　对整数数组给出快速排序的递归算法。

分析：只需要按照前面的分析写出程序即可。

代码：

```
#include<stdio.h>
void QkSort(int * s,int i,int j);
int QkPass(int * s,int i,int j);
int main()
{
    int s[]={50,39,64,90,72,12,29};
    //在屏幕上显示数组
```

```
    Show(s,7);
    //排序并显示排序后的结果
    printf("数组排序...\n");
    QkSort(s,0,6);
    //显示排序后的结果
    Show(s,7);
    return 0;
}
//在屏幕上显示数组
void Show(int * s,int n)
{
    for(int k=0;k<n;k++)
        printf("%d ",* (s+k));
    printf("\n");
}

//快速排序。s是待排序的数组,i和j指示了对数组从i到j处的数据进行排序,i<j
void QkSort(int * s,int i,int j)
{
    if(i < j)
    {
        /* 对数组A调用QkPass函数进行一趟快速排序。
        i和j指示了排序的起始和终了位置(下标),返回值指示了一趟排序后的分割点 * /
        int k = QkPass(s,i,j)
        //对前一部分继续快速排序,递归调用
        QkSort(s,i,k - 1)
        //对后一部分快速排序
        QkSort(s,k+1,j)
    }
}
//一趟快速排序的函数,对数组A从i到j快速排序,并返回分割点
int QkPass(int * s,int i,int j)
{
    //存储关键字
    int key s[i];
    while(i < j)
    {
        while(i < j && s[j] >= key)
            j--;                        //从后向前搜寻比key小的值
        s[i] = s[j]                     //找到后放入A(i)
        while(i < j && s[i] <= key)
            i++;                        //从前向后搜寻比key大的值
        s[j] = s[i]                     //找到后放入A(j)
    }
```

```
    //循环结束时,i=j,放入 key 值,并返回 i
    s[i] = key
    return i
}
```

快速排序每次将待排序数组分为两个部分。在理想状况下,每一次都将待排序数组划分成等长的两个部分,则需要 $\log_2 n$ 次划分;而在最坏情况下,即数组已经有序或大致有序的情况下,每次划分只能减少一个元素,快速排序将退化为冒泡排序,所以快速排序时间复杂度下界为 $O(n\log n)$,最坏情况为 $O(n^2)$。在实际应用中,快速排序的平均时间复杂度为 $O(n\log n)$。

快速排序的最坏情况基于每次划分对关键数据的选择。基本的快速排序选取第一个元素作为关键数据。这样在数组已经有序的情况下,每次划分将得到最坏的结果。一种比较常见的优化方法是随机化算法,即随机选取一个元素作为关键数据。这种情况下虽然最坏情况仍然是 $O(n^2)$,但最坏情况不再依赖于输入数据,而是由于随机函数取值不佳。实际上,随机化快速排序得到理论上最坏情况的可能性仅为 $1/(2^n)$。所以随机化快速排序可以对于绝大多数输入数据达到 $O(n\log n)$的期望时间复杂度。

*7.6　常用算法简介

对于计算机科学来说,算法的概念是至关重要的。例如,在一个大型软件系统的开发中,设计出有效的算法将起决定性的作用。通俗地讲,算法是指解决问题的一种方法或一个过程。程序与算法不同,程序是算法用某种程序设计语言的具体实现。

7.6.1　递归与分治

任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小,解题所需的计算时间往往也越少,从而也较容易处理。例如,对于 n 个元素的排序问题,当 $n=1$ 时,不需任何计算;$n=2$ 时,只要作一次比较即可排好序;$n=3$ 时,只要作两次比较即可;而当 n 较大时,问题就不那么容易处理了。要想直接解决一个较大的问题,有时是相当困难的。分治法的设计思想是,将一个难以直接解决的大问题分割成一些规模较小的相同的问题,各个击破,分而治之。如果原问题可分割成 k 个子问题,$1<k\leqslant n$,且这些子问题都可解,利用这些子问题的解求出原问题的解,那么这种分治法就是可行的。由分治法产生的问题往往是原问题的较小模式,这就为使用递归技术提供了方便。在这种情况下,反复应用分治手段,可以使子问题与原问题类型一致而其规模却不断缩小,最终使子问题缩小到很容易求解的规模。这样,就自然导致递归算法的产生。

一个直接或间接地调用自身的算法称为递归算法,一个使用函数自身给出定义的函数称为递归函数。使用递归往往使函数的定义和算法的描述简洁。

分治的基本思想为，对于一个规模为 n 的问题，若该问题可以较容易地解决（比如规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。

分治法所能解决的问题一般具有以下几个特征：

（1）该问题的规模缩小到一定的程度就可以很容易地解决。

（2）该问题可以分解为若干个规模较小的相同问题。

（3）利用该问题分解出的子问题的解可以合并为该问题的解。

（4）该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

上述的第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加。第二条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用。第三条特征是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑用贪心法或动态规划法。第四条特征涉及分治法的效率问题，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好。

根据分治法的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才较适当？这些问题没有确定的答案。但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。许多问题可以取 $k=2$。这种使子问题规模大致相等的做法是出自一种平衡（balancing）子问题的思想，它几乎总是比子问题规模不等的做法要好。

折半查找则是分治策略的典型的例子。折半查找函数 BinSearch 中的 while 循环是决定算法快慢的关键。很容易看出，每执行一次算法的 while 循环，待搜索数组的大小减少一半。因此，在最坏的情况下，while 循环被执行了 $O(\log n)$ 次。循环体内运算需要 $O(1)$ 时间，因此整个算法在最坏的情况下的计算时间复杂度为 $O(\log n)$。

快速排序则是基于分治策略的排序算法。回顾快速排序的过程，不难看出，快速排序是按分治法的 3 个步骤——分解、递归求解与合并来完成的。快速排序的运行时间按与划分是否对称有关。最坏的情况发生在划分过程产生的两个区域分别包含 $n-1$ 个元素和 1 个元素的时候。此时其时间复杂度为 $O(n^2)$。在最好的情况下，每次划分所取得基准都恰好是中值，此时的时间复杂度为 $O(n\log n)$。

7.6.2 动态规划

动态规划（dynamic programming）是运筹学的一个分支，是求解决策过程（decision process）最优化的数学方法。20 世纪 50 年代初美国数学家 R. E. Bellman 等人在研究多阶段决策过程（multistep decision process）的优化问题时，提出了著名的最优化原理（principle of optimality），把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关

系，逐个求解，创立了解决这类过程优化问题的新方法——动态规划。他在1957年出版了名著 *Dynamic Programming*，这是该领域的第一部著作。

动态规划问世以来，在经济管理、生产调度、工程技术和最优控制等方面得到了广泛的应用。例如最短路线、库存管理、资源分配、设备更新、排序、装载等问题，用动态规划方法比用其他方法求解更为方便。

动态规划程序设计是对解最优化问题的一种途径和一种方法，而不是一种特殊算法，不像前面所述的那些搜索或数值计算那样，具有一个标准的数学表达式和明确清晰的解题方法。动态规划程序设计往往是针对一种最优化问题，由于各种问题的性质不同，确定最优解的条件也互不相同，因而动态规划的设计方法对不同的问题有各具特色的解题方法，而不存在一种万能的动态规划算法可以解决各类最优化问题。因此读者在学习时，除了要对基本概念和方法正确理解外，必须具体问题具体分析，以丰富的想象力去建立模型，用创造性的技巧去求解。也可以通过对若干有代表性的问题的动态规划算法进行分析、讨论，逐渐学会并掌握这一设计方法。

动态规划算法通常用于求解具有某种最优性质的问题。在这类问题中，可能会有许多可行解。每一个解都对应于一个值，我们希望找到具有最优值的解。动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合用动态规划求解的问题，经分解得到的子问题往往不是互相独立的。若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次。如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。可以用一个表来记录所有已解的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果填入表中。这就是动态规划法的基本思路。具体的动态规划算法多种多样，但它们具有相同的填表格式。

任何思想方法都有一定的局限性，超出了特定条件，它就失去了作用。同样，动态规划也并不是万能的。适用动态规划的问题必须满足最优化原理和无后效性。

最优化原理也就是最优子结构性质：一个最优化策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理时又称其具有最优子结构性质。

无后效性是指将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态来决策。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。

子问题的重叠性。动态规划将原来具有指数级复杂度的搜索算法改进成了具有多项式时间的算法。其中的关键在于解决冗余，这是动态规划算法的根本目的。动态规划实质上是一种以空间换时间的技术，它在实现的过程中不得不存储产生过程中的各种状态，所以它的空间复杂度要大于其他的算法。

【例 7-10】 0-1背包问题。给定 n 种物品和一个背包。物品 i 的重量是 w_i，其价值

为 v_i，背包的容量为 c。问应如何选择装入背包中的物品，使得装入背包中物品的总价值最大？在选择装入背包的物品时，对每种物品 i 只有两种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i。因此，该问题称为 0-1 背包问题。

分析：首先，该问题具有最优子结构。也就是说，假设 $y_1, y_2, \cdots, y_n$ 是装入背包的一种最优方案。那么，$y_1, y_2, \cdots, y_{n-1}$ 也是装入到容量为 $c-w_n$ 背包中的一种最优方案。其次，不难看出，该问题具有递归的性质。

假设函数 $f(n,c)$ 表示将 n 件物品装入容量为 c 的背包可获得的最大价值。则对于第一件物品而言，具有两种选择：装入或者不装入。若选择装入，则剩余 $n-1$ 件物品装入容量为 $c-w_1$ 的背包中，构成最优子结构的解。反之，则将 $n-1$ 件物品装入容量为 c 背包中。显然：

$$f(n,c) = \mathrm{MAX}(f(n-1,c), f(n-1,c-w_1)+v_1)$$

递归求解，即可获得最优解。

而递归的边界条件是：若只有一件物品，当背包还有容量时，则装入，否则不装入。

然而，这样做的效率是低下的。动态规划的另一个特点就是记录曾经计算过的值。这样，在以后需要的时候，不需要重新计算，而是可以直接得到。在此使用二维数组 m 来记录，$m[i,j]$ 表示将第 $i, i+1, \cdots, n$ 个物品装入到容量为 j 的背包中可获得的最大价值。这样，求解背包问题，则是对 m 数组的填充。

根据以上的分析，假设有如下过程：

```
void KnapSack(int v[],int w[],int c,int n,int m[][N])
/* 程序的主要功能是对 m数组的填充。
    m(i,j)表示将第 i,i+1,…,n个物品装入到容量为 j的背包中可获得的最大价值 * /
//当 i=n时,只有一件物品有可能放入
   int jMax =w[n - 1] - 1< c?w[n- 1]- 1:c;
   int j;
   for(j =0; j<=jMax;j++)
        m[n][ j] =0;
   for(j =w[n - 1];j<=c To c
        m[n][ j] =v[n - 1]
   //自底向上推演,计算 n-1到 2的情形
   for(int i =n - 1;i>=2;i--)
{
     jMax =w[n - 1] - 1< c?w[n- 1]- 1:c;
     for(j =0; j<=jMax;j++)
          m[i][j] =m[i+1][j];
     for(j =w[i - 1]; j<=c;j++)
          m[i][j] =m[i+1][j]>m[i+1][j -w[i - 1]]+v[i - 1]? m[i+1][j]:
                m[i+1][j -w[i - 1]]+v[i - 1];
}
//考虑第一件物品选取或不选取的情形
```

```
    m[1][c] =m[2][c]
    If(c >=w[0] )
        m[1][c] =m[1][c]>m[2][c - w[0]+ v[0]]? m[1][c]: m[2][c - w[0]+ v[0]];
}
```

要注意的是,该过程计算后,在 $m[1][c]$中的值就是背包问题的最优值。具体该选取哪些物品其实也记录在 m 数组中了,可以用过程 TraceBack 构造如下。如果 $m[1][c]=m[2][c]$,则第一件物品不选取,$x1=0$,否则 $x1=1$。当 $x1=0$ 时,由 $m[2][c]$继续构造最优解,当 $x1=1$ 时,由 $m[2][c-w0]$继续构造最优解:

```
void TraceBack(int m[][N],int w[],int c,int n,int x[])
{
    for(int i =1; i<n;i++ )
    {
        if(m[i][c] =m[i+ 1][c])
            x[i] =0;
        else
        {
            x[i] =1;
            c=c - w[i - 1];
        }
        if(m[n][c] > 0)
            x[n] =1;
        else
            x[n] =0;
    }
}
```

7.6.3 贪心算法

贪心算法(又称贪婪算法)是指,在对问题求解时,总是做出在当前看来是最好的选择。也就是说,不从整体最优上加以考虑,它所做出的仅是在某种意义上的局部最优解。贪心算法不是对所有问题都能得到整体最优解,但对范围相当广泛的许多问题能产生整体最优解或者是整体最优解的近似解。

当一个问题具有最优子结构性质时,一般会用动态规划法求解。但有时会有更简便的算法。下面看一个找硬币的例子。假设有 4 种硬币,它们的面值分别为二角五分、一角、五分和一分。现在要找给某顾客六角三分钱。这时,我们会不假思索地拿出 2 个二角五分硬币、1 个一角的硬币和 3 个一分的硬币交给顾客。这种找硬币方法与其他的找法相比,拿出的硬币个数是最少的。这里使用了这样的找硬币算法:首先选出一个面值不超过六角三分的最大硬币,即二角五分;然后从六角三分中减去二角五分,剩下三角八分;再拿出一个面值不超过三角八分的最大硬币,即又一个二角五分,如此一直做下去。这个找硬币方法实际上就是贪心算法。顾名思义,贪心算法总是作出在当前看来是最好的选

择。也就是说贪心算法并不从整体最优上加以考虑,它所作出的选择只是在某种意义上的局部最优。当然,我们希望贪心算法得到的最终结果也是整体最优的。上面所说的找硬币算法得到的结果就是一个整体最优解。找硬币问题本身具有最优子结构性质,它可以用动态规划算法求解。但我们看到,用贪心算法更简单、更直接且解题效率更高。这利用了问题本身的一些特性。例如,上述找硬币的算法利用了硬币面值的特殊性。如果硬币的面值改为一分、五分和一角一分 3 种,而要找给顾客的是一角五分钱。还用贪心算法,我们将找给顾客 1 个一角一分的硬币和 4 个一分的硬币。然而 3 个五分的硬币显然是最好的找法。虽然贪心算法不是对所有题都能得到整体最优解,但对范围相当广的许多问题能产生整体最优解。

贪心算法通过一系列的选择来得到一个问题的解。它所作的每一个选择都是当前状态下某种意义的最好选择,即贪心选择。希望通过每次所作的贪心选择导致最终结果是问题的一个最优解。这种启发式的策略并不总能奏效,然而在许多情况下确实能达到预期的目的。

对于一个具体的问题,我们怎么知道是否可用贪心算法来解此问题,以及能否得到问题的一个最优解呢?这个问题很难给予肯定的回答。但是,从许多可以用贪心算法求解的问题中可以看到它们一般具有两个重要的性质:贪心选择性质和最优子结构性质。

1. 贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择,即贪心选择来达到。这是贪心算法可行的第一个基本要素,也是贪心算法与动态规划算法的主要区别。在动态规划算法中,每步所作的选择往往依赖于相关子问题的解。因而只有在解出相关子问题后才能作出选择。而在贪心算法中,仅在当前状态下作出最好选择,即局部最优选择。然后再去解作出这个选择后产生的相应的子问题。贪心算法所作的贪心选择可以依赖于以往所作过的选择,但绝不依赖于将来所作的选择,也不依赖于子问题的解。正是由于这种差别,动态规划算法通常以自底向上的方式解各子问题,而贪心算法则通常以自顶向下的方式进行。以迭代的方式作出相继的贪心选择,每作一次贪心选择就将所求问题简化为一个规模更小的子问题。

对于一个具体问题,要确定它是否具有贪心选择性质,必须证明每一步所作的贪心选择最终导致问题的一个整体最优解。首先考察问题的一个整体最优解,并证明可修改这个最优解,使其以贪心选择开始。而且作了贪心选择后,原问题简化为一个规模更小的类似子问题。然后,用数学归纳法证明,通过每一步作贪心选择,最终可得到问题的一个整体最优解。其中,证明贪心选择后的问题简化为规模更小的类似子问题的关键在于利用该问题的最优子结构性质。

2. 最优子结构性质

当一个问题的最优解包含它的子问题的最优解时,称此问题具有最优子结构性质。问题所具有的这个性质是该问题可用动态规划算法或贪心算法求解的一个关键特征。

7.6.4 回溯法

回溯法有“通用的解题法”之称。用它可以系统地搜索一个问题的所有解或任一解。回溯法是一个既带有系统性又带有跳跃性的搜索算法。它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。算法搜索至解空间树的任一结点时，总是先判断该结点是否肯定不包含问题的解。如果肯定不包含，则跳过对以该结点为根的子树的系统搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先的策略进行搜索。回溯法在用来求问题的所有解时，要回溯到根，且根结点的所有子树都已被搜索遍才结束。而回溯法在用来求问题的任一解时，只要搜索到问题的一个解就可结束。这种以深度优先的方式系统地搜索问题的解的算法称为回溯法，它适用于解一些组合数较大的问题。

应用回溯法解问题时，首先应明确定义问题的解空间。问题的解空间应至少包含问题的一个（最优）解。例如，对于有 n 种可选择物品的 0-1 背包问题，其解空间由长度为 n 的 0-1 向量构成。该解空间包含了对变量的所有可能的 0-1 赋值。当 $n=3$ 时，其解空间是

$$\{(0,0,0),(0,1,0),(0,0,1),(1,0,0),(0,1,1),(1,0,1),(1,1,0),(1,1,1)\}$$

定义了问题的解空间后，还应将解空间很好地组织起来，使得用回溯法能方便地搜索整个解空间。通常将解空间组织成树或图的形式。

例如，对于 $n=3$ 时的 0-1 背包问题，其解空间用一棵完全二叉树表示，如图 7-11 所示。

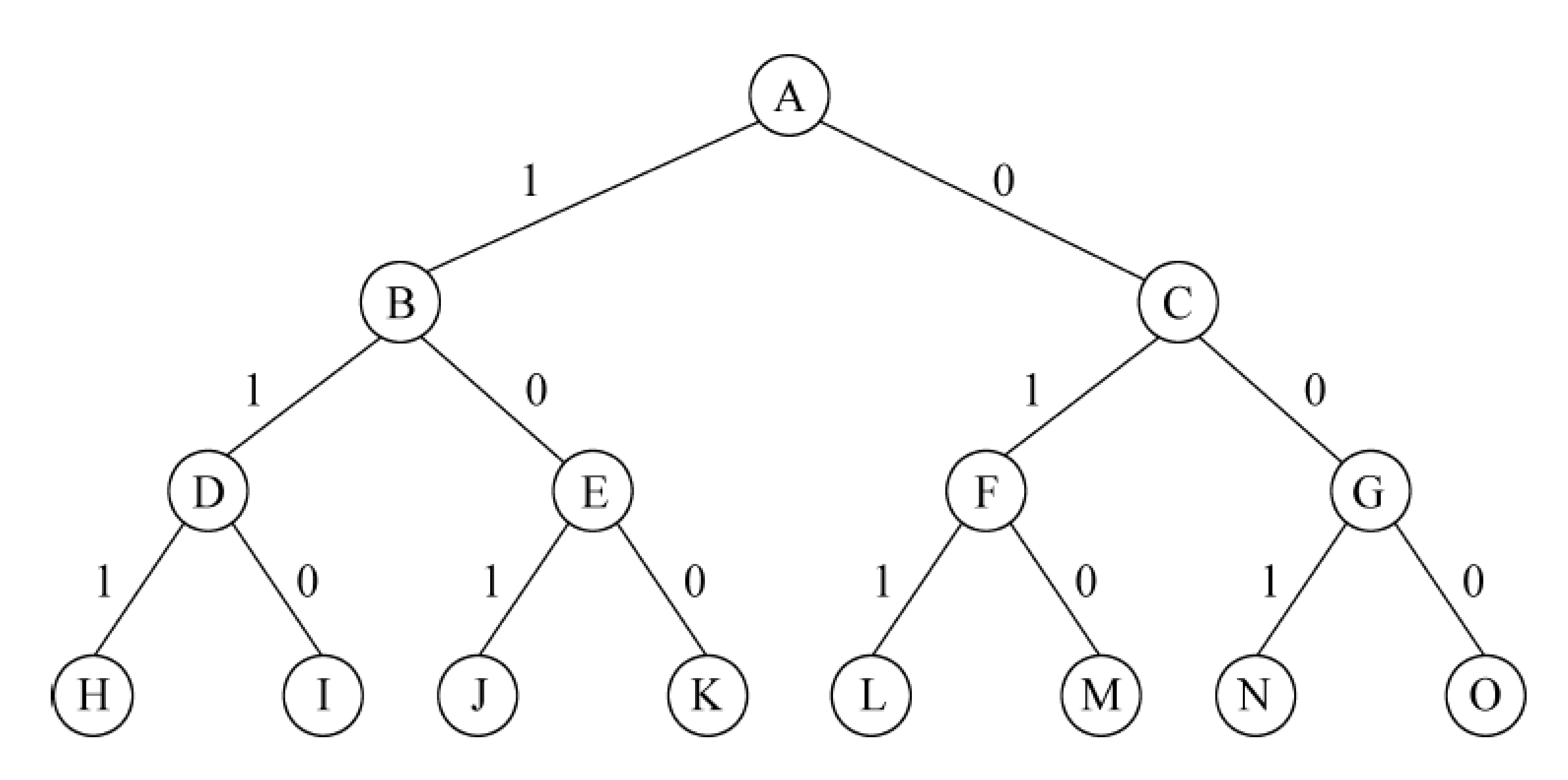


图 7-11　0-1 背包问题的解空间树

解空间树的第 i 层到第 $i+1$ 层边上的标号给出了变量的值。从树根到叶的任一条路径表示解空间的一个元素。例如，从根结点到结点 H 的路径对应于解空间的元素 (1,1,1)。

确定了解空间的组织结构后，回溯法就从开始结点（根结点）出发，以深度优先的方式搜索整个解空间。这个开始结点就成为一个活结点，同时也成为当前的扩展结点。在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为一个新的活结点，并成为当前扩展结点。如果在当前的扩展结点处不能再向纵深方向移动，则当前的扩展

结点就成为死结点。换句话说,这个结点不再是一个活结点。此时,应往回移动(回溯)至最近的一个活结点处,并使这个活结点成为当前的扩展结点。回溯法即以这种工作方式递归地在解空间中搜索,直至找到所要求的解或解空间中已无活结点时为止。

例如,对于 $n=3$ 时的 0-1 背包问题,考虑下面的具体实例:$w=[16,15,15]$,$p=[45,25,25]$,$c=30$。从图 7-11 的根结点开始搜索其解空间。开始时根结点是唯一的活结点,也是当前的扩展结点。在这个扩展结点处,可以沿纵深方向移至结点 B 或结点 C。假设选择先移至结点 B。此时,结点 A 和结点 B 是活结点,结点 B 成为当前扩展结点。由于选取了 w_1,故在结点 B 处剩余背包容量 $r=14$,获取的价值为 45。从结点 B 处,可以移至结点 D 或 E,由于移至结点 D 至少需要 $w_2=15$ 的背包容量,而现在仅有的背包容量是 14,故移至结点 D 导致一个不可行解。而搜索至结点 E 不需要背包容量,因而是可行的。从而选择移至结点 E。此时,E 成为新的扩展结点,结点 A、B 和 E 是活结点。在结点 E 处,$r=14$,获取的价值为 45。从结点 E 处,可以向纵深移至结点 J 或 K。移至结点 J 导致一个不可行解,而移向结点 K 是可行的,于是移向结点 K,它成为一个新的扩展结点。由于结点 K 是一个叶结点,故得到一个可行解。这个解相应的价值为 45。x_i 的取值由根结点到叶结点 K 的路径所唯一确定,即 $x=(1,0,0)$。由于在结点 K 处已不能再向纵深扩展,所以结点 K 成为死结点。于是返回到结点 E 处。此时在结点 E 处也没有可扩展的结点,它也成为死结点。

接下来又返回到结点 B 处。结点 B 同样也成为死结点,从而结点 A 再次成为当前扩展结点。结点 A 还可继续扩展,从而到达结点 C。此时,$r=30$,获取的价值为 0。从结点 C 可移向结点 F 或 G。假设移至结点 F,它成为新的扩展结点。结点 A、C 和 F 是活结点。在结点 F 处,$r=15$,获取的价值为 25。从结点 F 向纵深移至结点 L 处,此时,$r=0$,获取价值为 50。由于 L 是一个叶结点,而且是迄今为止找到的获取价值最高的可行解,因此记录这个可行解。结点 L 不可扩展,因此又返回到结点 F 处。按此方式继续搜索,可搜索遍整个解空间。搜索结束后找到的最好解是相应 0-1 背包问题的最优解。

习　题

1. 编写程序,使用冒泡排序对 10 个整数排序。
2. 改进冒泡排序程序,使其当数据已经有序时直接结束排序的过程。
3. 编写程序,使用快速排序对 10 个整数排序。
4. 改进快速排序程序,随机选取关键数据。
5. 编写程序,使用冒泡排序对电话号码簿按人名的字典顺序排序。
6. 使用顺序查找,对第 5 题的数据查找一个人名是否在电话簿中。
7. 使用折半查找,对排序后的电话簿(第 5 题)进行查找。
8. 给定 K 个整数的序列$\{N_1,N_2,\cdots,N_K\}$,其任意连续子序列可表示为$\{N_i,N_{i+1},\cdots,N_j\}$,其中 $1\leqslant i\leqslant j\leqslant K$。求最大连续子序列,即所有连续子序列中元素和最大的一个,例如给定序列$\{-2,11,-4,13,-5,-2\}$,其最大连续子序列为$\{11,-4,13\}$,最大和

为 20。

9. 数论中有许多猜想尚未解决，其中有一个被称为“角谷猜想”的问题，这个问题是这样描述的：任何一个大于 1 的自然数，如果是奇数，则乘以 3 再加 1；如果是偶数，则除以 2；得出的结果继续按照前面的规则进行运算，最后必定得到 1。请编写一个程序验证。

10. 某部队进行新兵队列训练，将新兵从 1 开始按顺序依次编号，并排成一行横队。训练的规则如下：从头开始 1 至 2 报数，凡报到 2 的出列，剩下的向小序号方向靠拢；再从头开始进行 1 至 3 报数，凡报到 3 的出列，剩下的向小序号方向靠拢；继续从头开始进行 1 至 2 报数；以后从头开始轮流进行 1 至 2 报数、1 至 3 报数，直到剩下的人数不超过 3 人为止。编写程序，输入数 N 为最开始的新兵人数（$20<N<6000$），输出剩下的新兵最初的编号。

11. 在医院打点滴（吊针）的时候，如果滴起来有规律：先是滴一滴，停一下；然后滴二滴，停一下；再滴三滴，停一下……现在有一个问题：这瓶盐水一共有 v 毫升，每一滴是 d 毫升，每一滴的速度是 1s（假设最后一滴不到 d 毫升，则花费的时间也算 1s），停一下的时间也是 1s，这瓶水什么时候能滴完呢？（设 $0<d<v<1000$）。



第8章 数据结构基础

引言

我们已经知道，用计算机解决一个具体问题时，首先要从具体问题抽象出一个适当的数学模型，然后设计一个解此数学模型的算法，最后编出程序，进行测试、调整直至得到最终解答。如果一个问题可以用数学的方程来描述，如人口增长可以用微分方程来描述，只需要输入相应的数据，然后通过计算机求解该方程即可。这一般称为数值计算。然而，很多问题是无法用数学方程来描述的，如计算机和人的对弈问题。这类非数值计算问题需要用其他的方式来描述和求解，其核心是数据结构和算法。

本节对常用的数据结构作了简单讲述，实现了线性表、队列和栈等常用的数据结构。

教学目的

- 掌握数据与数据结构的基本概念。
- 掌握线性表的基本结构及其上的运算。
- 掌握栈和队列的基本概念。
- 了解树和图的基本概念。

8.1 数据与数据结构

8.1.1 数据

什么叫数据？数据是描述客观事物的信息符号的集合，这些信息符号能被输入到计算机中存储起来，又能被程序处理、输出。事实上，数据这个概念本身是随着计算机的发展而不断扩展的概念。在计算机发展的初期由于计算机主要用于数值计算，数据指的就是整数、实数等数值；在计算机用于文字处理时，数据指的就是由英文字母和汉字组成的字符串；随着计算机硬件和软件技术的不断发展，扩大了计算机的应用领域，诸如表格、图形、图像、声音等也属于数据的范畴。目前非数值问题的处理占用90%以上的计算机

时间。

数据类型是程序设计中的概念，程序中的数据都属于某个特殊的数据类型，它是指具有相同特性的数据的集合。数据类型决定了数据的性质，如取值范围、操作运算等。常用的数据类型有整型、浮点型、字符型等。数据类型还决定了数据在内存中所占空间的大小，如字符型占 1 个字节，而长整型一般占 4 个字节等。

对于复杂一些的数据，仅用数据类型无法完整地描述。例如，在图 8-1 中，表示教师得分要描述教师的姓名、各项得分，这时需要用到数据元素的概念。数据元素中可能用到多个数据类型（称为数据项），共同描述一个客体，如教师。数据元素有时也被称为记录或结点。在程序设计中，前面所说的数据类型又被称为基本数据类型，由基本数据类型组成的数据元素的定义被称为构造数据类型（结构和类都属于后者）。

教师得分登记表

姓名	教学得分	科研得分	其他得分	合计
张力	35	34	11	80
王五	36	35	12	83
⋮	⋮	⋮	⋮	⋮

教师得分登记表的数据元素是姓名、教学得分、科研得分、其他得分、合计，也就是说每个数据元素由姓名、教学得分、科研得分、其他得分、合计5个数据项组成。这5个数据项含义明确，若再细分就无明确独立的含义，属于基本数据类型(字符型和整型或浮点型)。

图 8-1　教师得分登记表

8.1.2　数据结构

计算机的处理效率与数据的组织形式和存储结构密切相关。这类似于人们所用的词典和字典等工具书，它们都是按字母或拼音字母的顺序组织排列词条，这样人们查阅工具书的速度较快。假如词条不是按字母顺序组织排列，而是按任意顺序组织排列，那么查词速度一定很慢。因此，很有必要研究数据的组织形式和存储结构。另外在当今网络世界中传递数据更加依赖于数据的组织形式和存储结构。

什么是数据结构？数据结构在计算机科学界至今没有标准的定义。各人根据各自的理解而有不同的表述方法。

Sartaj Sahni 在他的《数据结构、算法与应用》一书中称：“数据结构是数据对象，以及存在于该对象的实例和组成实例的数据元素之间的各种联系。这些联系可以通过定义相关的函数来给出。”他将数据对象（data object）定义为“一个数据对象是实例或值的集合”。

Clifford A. Shaffer 在《数据结构与算法分析》一书中的定义是：“数据结构是 ADT（Abstract Data Type，抽象数据类型 ）的物理实现”。

Lobert L. Kruse 在《数据结构与程序设计》一书中，将一个数据结构的设计过程分成抽象层、数据结构层和实现层。其中，抽象层是指抽象数据类型层，它讨论数据的逻辑结

构及其运算,数据结构层和实现层讨论一个数据结构的表示和在计算机内的存储细节以及运算的实现。

由此可见,在任何问题中,构成数据的数据元素并不是孤立存在的,它们之间存在着一定的关系以表达不同的事物及事物之间的联系。所以,简单地说,数据结构就是研究数据及数据元素之间关系的一门学科,它包括以下3个方面的内容:

- 数据的逻辑结构。
- 数据的存储结构。
- 数据的运算(即数据的处理操作)。

一般认为,一个数据结构是由数据元素依据某种逻辑联系组织起来的。对数据元素间逻辑关系的描述称为数据的逻辑结构。数据必须在计算机内存储,数据的存储结构是数据结构的实现形式,是其在计算机内的表示。此外,讨论一个数据结构必须同时讨论在该类数据上执行的运算才有意义。

1. 数据的逻辑结构

数据的逻辑结构就是数据元素之间的逻辑关系。这里,我们对数据所描述的客观事物本身的属性意义不感兴趣,只关心它们的结构及关系。将那些在结构形式上相同的数据抽象成某一数据结构,比如线性表、树和图等。

根据数据元素之间关系的不同特性,数据结构又可分为以下四大类(图8-2):

(1) 集合。数据元素之间的关系只有“是否属于同一个集合”。

(2) 线性结构。数据元素之间存在线性关系,即最多只有一个前趋和后继元素。

(3) 树。数据元素之间存在层次关系,即最多有一个前趋和多个后继元素。

(4) 图。数据元素之间的关系为多对多的关系。

其中树和图又被统称为非线性数据结构。

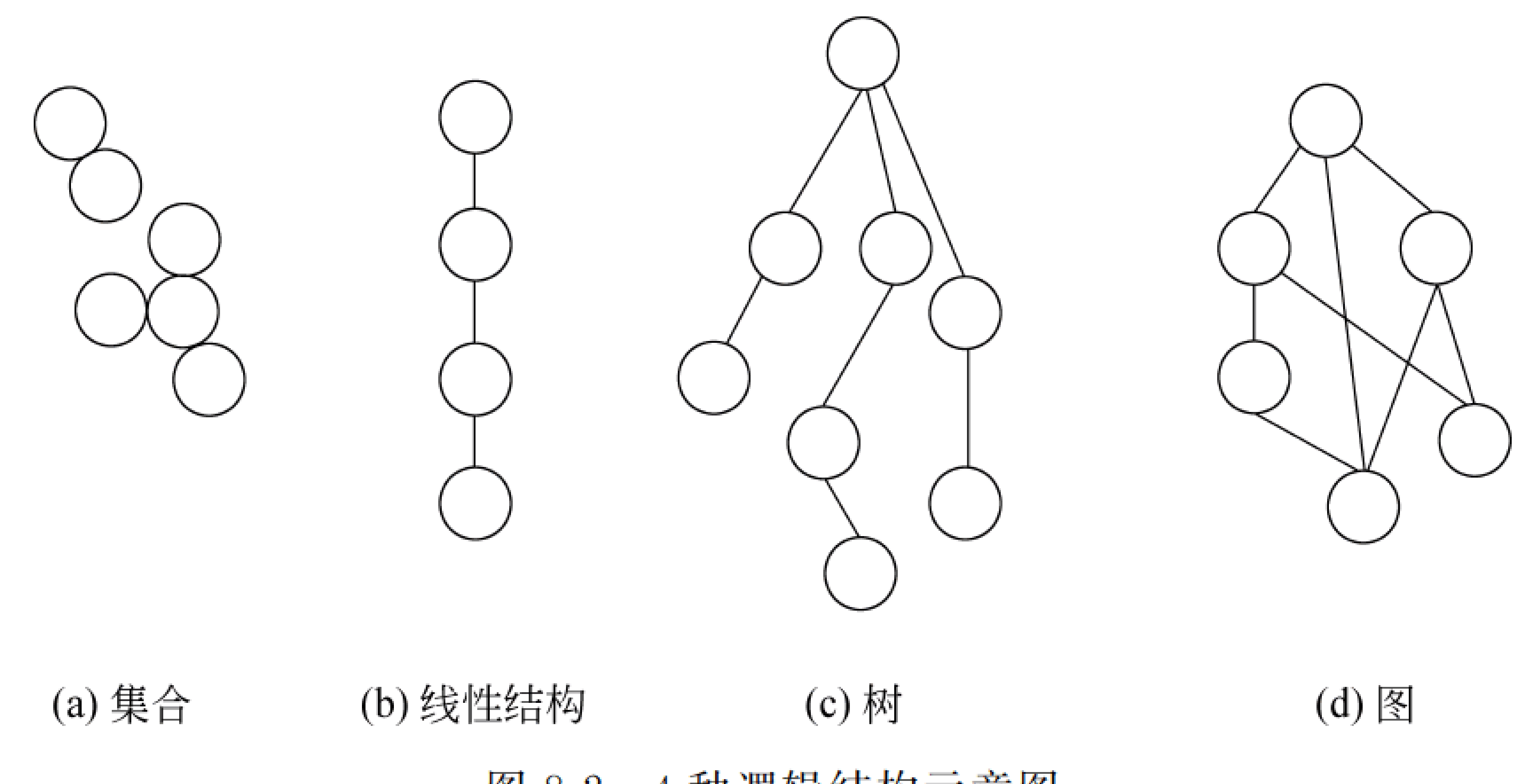


图8-2 4种逻辑结构示意图

2. 数据的存储结构

数据的逻辑结构是从逻辑上来描述数据元素之间的关系的,是独立于计算机的。然而讨论数据结构的目的是为了在计算机中实现对它的处理。因此还需要研究数据元素和

数据元素之间的关系如何在计算机中表示，这就是数据的存储结构。又称数据的映像。

计算机的存储器是由很多个存储单位组成的，每个存储单元有唯一的地址。数据的存储结构要讨论的就是数据结构在计算机存储器上的存储映像方法。根据数据结构的形式定义，数据结构在存储器上的映像不仅包括数据元素集合如何存储映像，而且还包括数据元素之间的关系如何存储映像。

一般来说，数据在存储器中的存储有 4 种基本的映像方法。

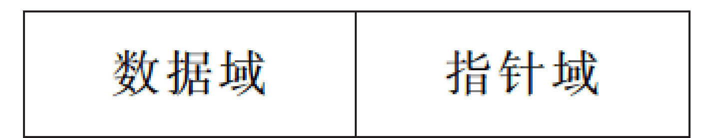
1）顺序存储结构

顺序存储结构是把数据元素按某种顺序放在一块连续的存储空间中，其特点是借助数据元素在存储器中的相对位置来表示数据元素之间的关系。顺序存储的问题是，如果元素集合很大，则可能找不到一块很大的连续空间来存放。

2）链式存储结构

有时往往存在这样一些情况：存储器中没有足够大的连续可用空间，只有不相邻的零碎小块存储空间；另一种情况是在事前申请一段连续空间时，因无法预计所需存储空间的大小，需要临时增加空间。所有这些情况说明，要得到一块合适的连续存储空间并非易事，即在这种情况下顺序存储结构无法实现。

链式存储结构的特点就是将存放每个数据元素的结点分为两部分：一部分存放数据元素（称为数据域）；另一部分存放指示存储地址的指针（称为指针域），借助指针表示数据元素之间的关系。结点的结构如下：



链式存储结构可用一组任意的存储单元来存储数据元素，这组存储单元可以是连续的，也可以是不连续的。链式存储因为有指针域，增加了额外的存储开销，并且实现上也较为麻烦，但大大增加了数据结构的灵活性。

3）索引存储结构

在线性表中，数据元素可以排成一个序列：$R_1, R_2, R_3, \cdots, R_n$，每个数据元素 R_i 在序列里都有对应的位置码 i，这就是元素的索引号。索引存储结构就是通过数据元素的索引号 i 来确定数据元素 R_i 的存储地址。一般索引存储结构有两种实现方法：一是建立附加的索引表，索引表里第 i 项的值就是第 i 个元素的存储地址；二是当每个元素所占单元数都相等时，可用位置码 i 的线性函数值来确定元素对应的存储地址。

4）散列存储结构

这种存储方法就是在数据元素与其在存储器上的存储位置之间建立一个映像关系 F。根据这个映像关系 F，已知某数据元素就可以得到它的存储地址。即 $D=F(E)$，这里 E 是要存放的数据元素，D 是该数据元素的存储位置。可见，这种存储结构的关键是设计这个函数 F，但函数 F 不可能解决数据存储中的所有问题，还应有一套意外事件的处理方法，它们共同实现数据的散列存储结构。哈希表是一种常见的散列存储结构。

3. 数据的运算

数据的运算是定义在数据逻辑结构上的操作，如插入、删除、查找、排序、遍历等。每

种数据结构都有一个运算的集合。

8.2 线 性 表

线性表是最基本、最简单也是最常用的一种数据结构。线性表中数据元素之间的关系是一对一的关系，即除了第一个和最后一个数据元素之外，其他数据元素都是首尾相接的。线性表的逻辑结构简单，便于实现和操作，在实际应用中是广泛采用的一种数据结构。

8.2.1 线性表的逻辑结构及运算

线性表是一个线性结构，它是一个含有 $n \geqslant 0$ 个结点的有限序列，其中，有且仅有一个开始结点(第一个结点)没有前趋但有一个后继结点，有且仅有一个终端结点(最后一个结点)没有后继但有一个前趋结点，其他的结点都有且仅有一个前趋和一个后继结点。

一般地，一个线性表可以表示成一个线性序列：$k_1, k_2, \cdots, k_n$，其中 k_1 是开始结点，k_n 是终端结点。线性表具有以下一些基本性质：

- 数据元素的个数 n 定义为表的长度。当 $n=0$ 时称为空表，空表中无数据元素。
- 若表非空，则必存在唯一的一个开始结点。
- 必存在唯一的一个终端结点。
- 除最后一个元素之外，其余结点均有唯一的后继。
- 除第一个元素之外，其余结点均有唯一的前趋。
- 数据元素 $k_i (1 \leqslant i \leqslant n)$ 在不同情况下的具体含义不同，它可以是一个数，或者是一个符号，或者是更复杂的信息。虽然不同数据表的数据元素可以是各种各样的，但同一线性表的各数据元素必定具有相同的数据类型和长度。

【例 8-1】 线性表的例子。

- 某班学生的数学成绩(78,92,66,84,45,72,92)是一个线性表，每个数据元素是一个正整数，表长为 7。
- 一星期的 7 天的英文缩写词(SUN,MON,TUE,WED,THU,FRI,SAT)是一个线性表，表中数据元素是一个字符串，表长为 7。
- 某企业职工基本工资情况((张三,助工,3,543),(李四,高工,21,986),(王五,工程师,9,731))也是一个线性表，表中数据元素是由姓名、职称、工龄、基本工资 4 个数据项组成的一个记录(对象)，表长为 3。

线性表可以进行的常用基本操作有以下几种：

(1) 置空表。将线性表 L 的表长置为 0。

(2) 求表长。求出线性表 L 中数据元素的个数。

(3) 取表中元素。仅当 $1 \leqslant i \leqslant \mathrm{Length}(L)$ 时，取得线性表 L 中的第 i 个元素 k_i(或 k_i 的存储位置)，否则无意义。

(4) 取元素 k_i 的直接前趋。当 $2 \leqslant i \leqslant \text{Length}(L)$时,返回 k_i 的直接前趋 k_{i-1}。

(5) 取元素 k_i 的直接后继。当 $1 \leqslant i \leqslant \text{Length}(L)-1$ 时,返回 k_i 的直接后继 k_{i+1}。

(6) 定位。返回元素 x 在线性表 L 中的位置。若在 L 中有多个 x,则只返回第一个 x 的位置,若在 L 中不存在 x,则返回 0。

(7) 插入。在线性表 L 的第 i 个位置上插入元素 x,运算结果使得线性表的长度增加 1。

(8) 删除。删除线性表 L 的第 i 个位置上的元素 k_i,此运算的前提应是 $\text{Length}(L) \neq 0$,运算结果使得线性表的长度减 1。

对线性表还有一些更为复杂的操作,例如,将两个线性表合并成一个线性表,将一个线性表分解为 n 个线性表,对线性表中的元素按值的大小重新排列等。这些运算都可以通过上述 8 种基本运算的组合派生来实现。

8.2.2　顺序线性表

要使线性表成为计算机可以处理的对象,就必须把线性表的数据元素及数据元素之间的逻辑关系都存储到计算机的存储器中。线性表常用顺序方式和链表方式来存储。

线性表的顺序存储结构就是将线性表的每个数据元素按其逻辑次序依次存放在一组地址连续的存储单元里。由于逻辑上相邻的元素存放在内存的相邻单元中,所以线性表的逻辑关系蕴含在存储单元的物理位置相邻的关系中。也就是说,在顺序存储结构中,线性表的逻辑关系的存储是隐含的。

设线性表中每个元素占用 C 个存储单元,用 $\text{Loc}(k_i)$表示元素 k_i 的存储位置,则顺序存储结构的存储示意图如图 8-3 所示。

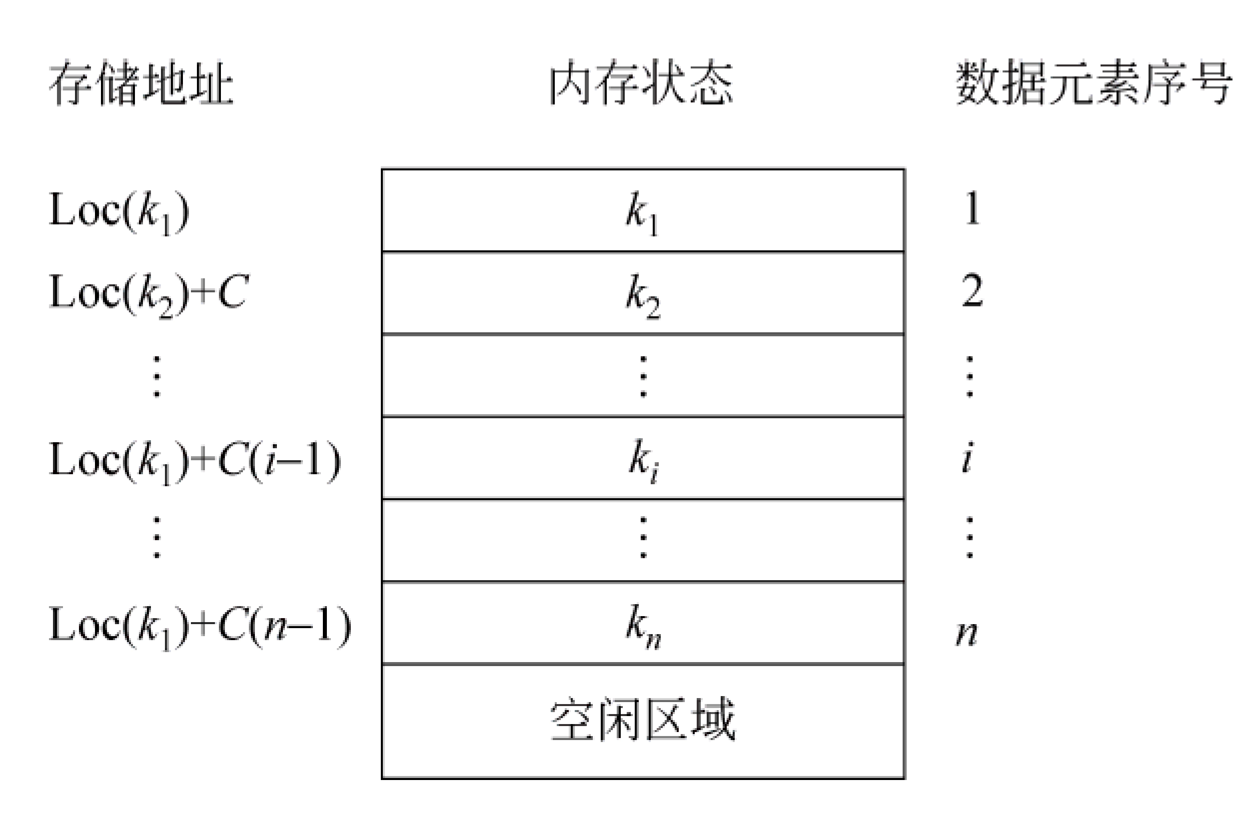


图 8-3　线性表的顺序存储结构示意图

从图 8-3 中可以看出,若已知线性表的第一个元素的存储位置是 $\text{Loc}(k_1)$,则第 i 个元素的存储位置为

$$\text{Loc}(k_i) = \text{Loc}(k_1) + C(i-1) \qquad 1 \leqslant i \leqslant n$$

可见,线性表中每个元素的存储地址是该元素在表中序号的线性函数。只要知道某元素在线性表中的序号,就可以确定其在内存中的存储位置。所以说,线性表的顺序存储结构是一种随机存取结构。

在 C 语言中，数组是在内存中连续分配的。所以数组就是一种线性结构。用数组来实现线性表，可以预先定义一个较大的数组，用来存放线性表中的元素。元素从数组的 0 位置存起，数组最后的一些位置是空闲的。

【例 8-2】 一个整数线性表的实现。用整型数组存储元素，实现线性表的基本操作。

分析：使用数组 list 来存储元素。list 的大小设为 MAX=1000。表长用 length 来表示，线性表中的每一个元素都是整数。这里，使用一个自定义的数据类型来描述线性表。该数据类型 ListType 包含一个数组用于存放数据。定义两个整数分别表示表长和表的最大容量。

程序：

```
#include <stdio.h>
#include <stdlib.h>
/* 本例设计一个元素类型为整数的线性表
 * 为了通用,此处将整数重定义为 DataType
 */
typedef int DataType;
//定义线性表的结构
typedef structList
{
    DataType* list;        //指向线性表的指针
    int length;            //表长
    int maxLength;         //表容量
}ListType;
//声明线性表具有的方法
ListType* CreateList(int length);               //创建一个长度为 length 的线性表
void DestroyList(ListType* pList);              //销毁线性表
void ClearList(ListType* pList);                //置空线性表
int IsEmptyList(ListType* pList);               //检测线性表是否为空
int GetListLength(ListType* pList);             //获取线性表长度
int GetListElement(ListType* pList, int n, DataType* data);
                                                //获取线性表中第 n 个元素
int FindElement(ListType* pList, int pos, DataType data);
                                                //从 pos 起查找 data 第一次出现的位置
int GetPriorElement(ListType* pList, int n, DataType* data);
                                                //获取第 n 个元素的前趋
int GetNextElement(ListType* pList, int n, DataType* data);
                                                //获取第 n 个元素的后继
int InsertToList(ListType* pList, int pos, DataType data);//将 data 插入到 pos 处
int DeleteFromList(ListType* pList, int pos);   //删除线性表上位置为 pos 的元素
void PrintList(ListType* pList);                //输出线性表
//主函数,创建一个线性表并测试
int main()
{
```

```
    const int MAXLENGTH = 1000;                    //假设最大容量为 1000
    //创建线性表
    ListType * sqList = CreateList(MAXLENGTH);
    //以下是对线性表的测试
    ClearList(sqList);                             //置表为空
    //插入 10 个元素并显示
    for (int i = 0; i < 10; ++ i)
        InsertToList(sqList, i, i + 1);
    //输出线性表
    PrintList(sqList);
    //在位置 5 插入 99 并显示
    InsertToList(sqList, 5, 99);
    printf("插入 99 后的线性表\n");
    PrintList(sqList);
    //删除第 8 个元素
    DeleteFromList(sqList, 8);
    printf("删除第 8 个元素后的线性表\n");
    PrintList(sqList);
    //显示第 3 个元素的前趋
    DataType data;
    if (GetPriorElement(sqList, 3, &data) > - 1);
        printf("第 3 个元素的前趋是%d\n", data);
    return 0;
}
//线性表方法实现
/* *
 * @brief  创建一个新的线性表
 * @param length  线性表的最大容量
 * @return  成功返回指向该表的指针,否则返回 NULL
 * /
ListType * CreateList(int length)
{
    ListType * sqList= (ListType * )malloc(sizeof(ListType));
    if (sqList ! = NULL)
    {
        //为线性表分配内存
        sqList-> list = (DataType * )malloc(sizeof(DataType) * length);
        //如果分配失败,返回 NULL
        if (sqList-> list == NULL)
            return NULL;
        //置为空表
        sqList-> length = 0;
        //最大长度
        sqList-> maxLength = length;
```

```
    }
    return sqList;
}
/**
 * @brief 销毁线性表
 * @param pList 指向需要销毁的线性表的指针
 */
void DestroyList(ListType* pList)
{
    free(pList->list);
}
/**
 * @brief 置空线性表
 * @param pList 指向需要置空线性表的指针
 */
void ClearList(ListType* pList)
{
    pList->length = 0;
}
/**
 * @brief 检测线性表是否为空
 * @param pList 指向线性表的指针
 * @return 如果线性表为空,返回 1;否则返回 0
 */
int IsEmptyList(ListType* pList)
{
  return pList->length == 0 ? 1 : 0;
}
/**
 * @brief 获取线性表长度
 * @param pList 指向线性表的指针
 * @return 线性表的长度
 */
int GetListLength(ListType* pList)
{
    return pList->length;
}
/**
 * @brief 获取线性表中第 n 个元素
 * @param pList 指向线性表的指针
 * @param n 要获取元素在线性表中的位置
 * @param data 获取成功,取得元素存放在 data 中
 * @return 获取成功返回 1,失败则返回 0
 */
```

```
int GetListElement(ListType* pList, int n, DataType* data)
{
    if (n<0 || n>pList->length - 1)
        return 0;
    * data = pList->list[n];
    return 1;
}
/* *
 * @brief 从 pos 起查找 data 第一次出现的位置
 * @param pList 指向线性表的指针
 * @param pos 查找的起始位置
 * @param data 要查找的元素
 * @return 找到则返回该位置,未找到返回-1
 * /
int FindElement(ListType* pList, int pos, DataType data)
{
    for (int n = pos; n < pList->length; ++n)
    {
        if (data == pList->list[n])
            return n;
    }
    return -1;
}
/* *
 * @brief 获取第 n 个元素的前趋
 * @param pList 指向线性表的指针
 * @param n n 的前趋
 * @param data 获取成功,取得元素存放在 data 中
 * @return 找到则返回前趋的位置(n-1),未找到返回-1
 * /
int GetPriorElement(ListType* pList, int n , DataType* data)
{
    if (n < 1 || n>pList->length-1)
        return -1;
    * data = pList->list[n - 1];
    return n-1;
}
/* *
 * @brief 获取第 n 个元素的后继
 * @param pList 指向线性表的指针
 * @param n n 的后继
 * @param data 获取成功,取得元素存放在 data 中
 * @return 找到则返回后继的位置(n+1),未找到返回-1
 * /
```

```
int GetNextElement(ListType* pList, int n,DataType* data)
{
    if (n<0 || n>pList->length - 2)
        return -1;
    *data = pList->list[n + 1];
    return n+1;
}
/**
 * @brief 将 data 插入到线性表的 pos 位置处
 * @param pList 指向线性表的指针
 * @param pos 插入的位置
 * @param data 要插入的元素存放在 data 中
 * @return 成功返回新的表长(原表长+1),失败返回-1
 */
int InsertToList(ListType* pList, int pos, DataType data)
{
    //如果插入的位置不正确或者线性表已满,则插入失败
    if (pos<0 || pos>pList->length || pList->length ==pList->maxLength)
        return -1;
    //从 pos 起,所有的元素向后移动一位
    for (int n =pList->length; n >pos; --n)
    {
        pList->list[n] =pList->list[n - 1];
    }
    //插入新的元素
    pList->list[pos] =data;
    //表长增加 1
    return ++pList->length;
}
/**
 * @brief 将 pos 位置处的元素删除
 * @param pList 指向线性表的指针
 * @param pos 删除元素的位置
 * @return 成功返回新的表长(原表长-1),失败返回-1
 */
int DeleteFromList(ListType* pList, int pos)
{
    if (pos<0 || pos>pList->length)
        return -1;
    //将 pos 后的元素向前移动一位
    for (int n =pos; n <pList->length - 1; ++n)
        pList->list[n] =pList->list[n + 1];
    return --pList->length;
}
```

```
/**
 * @brief  输出线性表
 * @param pList  指向线性表的指针
 */
void PrintList(ListType* pList)
{
    for (int n = 0; n < pList->length; ++n)
        printf("第%d项：%d\n", n, pList->list[n]);
}
```

需要注意的是，由于C语言中的数组下标是从0开始的，为方便起见，本程序实现的线性表默认的第一个元素下标是0。因此在第5个位置上插入，实际是在线性表的第6个位置上插入。也可以修改程序，使其和前面描述的线性表一致。程序运行的结果如下：

```
第 0项：1
第 1项：2
第 2项：3
第 3项：4
第 4项：5
第 5项：6
第 6项：7
第 7项：8
第 8项：9
第 9项：10
```

插入99后的线性表

```
第 0项：1
第 1项：2
第 2项：3
第 3项：4
第 4项：5
第 5项：99
第 6项：6
第 7项：7
第 8项：8
第 9项：9
第 10项：10
```

删除第8个元素后的线性表

```
第 0项：1
第 1项：2
第 2项：3
第 3项：4
第 4项：5
```

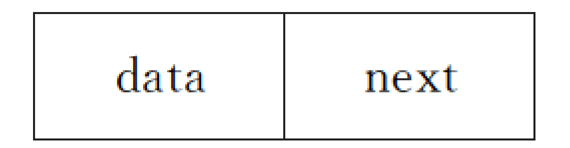
第 5项：99
第 6项：6
第 7项：7
第 8项：9
第 9项：10
第 3个元素的前趋是 3

8.2.3 链表

线性表的顺序存储结构是把整个线性表存放在一片连续的存储区域，其逻辑关系上相邻的两个元素在物理位置上也相邻，因此可以随机存取表中任一元素，每个元素的存储位置可用一个简单、直观的公式来表示。然而，如果需要对某一线性表中的元素频繁进行插入和删除操作，为了保持元素在存储区域的连续性，在插入元素时必须移动大量元素给新插入的元素“腾位置”，而在删除时，又必须移动大量后继元素“补缺”，因而在操作执行时要花大量时间去移动数据元素。此外顺序表类在创建时需要开辟较大的连续空间，而表中元素进进出出不可能一下子占满这块连续空间，所以存储空间的使用效率不高。

能否设计一种新的存储结构来弥补顺序存储结构的不足，尤其在元素插入、删除时无须改变已存储元素的位置？这就是下面将讨论的非顺序存储结构，又称链式存储结构。

链式结构用一组任意的存储区域存储线性表，此存储区域可以是连续的，也可以是分散的。这样，逻辑上相邻的元素在物理位置上就不一定是相邻的，为了能正确反映元素的逻辑次序，就必须在存储每个元素的同时，存储其直接后继（或直接前趋）的存储位置。因此在链式结构中每个元素都由两部分组成：存储数据元素的数据域（data），存储直接后继元素存储位置的指针域（next）。其存储结构示意如下：



在下面的讨论中将这样存储的每个数据元素称为结点。每个数据元素存储结构的定义如下：

```
typedef structNODE
{
    datatype   data;      //数据域
    Node *  next;         //指针域
}Node;
```

由于线性表每个元素都有唯一的后继（除了最尾元素），所以开辟指针域记录后继元素的地址。最尾元素的指针域为空，即为 NULL。数据元素本身的存储类型同顺序表一样定义成结构类型。线性表的非顺序存储结构如图 8-4 所示。

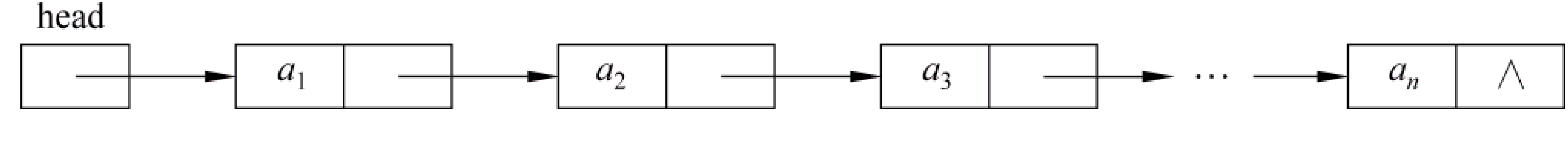


图 8-4　线性表的非顺序存储结构示意图

非顺序存储结构中,每个结点的存储既不是连续的也不是顺序的,而是散落在存储器的各个区域。通过指针将线性表中每个结点连接起来,指针就好像自行车链条一样。因此图 8-4 所示的线性表存储结构被称为单向链表,简称单链表。

单链表类的特点如下:

(1) 线性表中实际有多少元素就存储多少个结点。

(2) 元素存放可以不连续,其物理存放次序与逻辑次序不一定一致,换句话说,a_{i-1}可能存放在存储器的下半区,而 a_i可能存放在存储器的上半区。

(3) 线性表中元素的逻辑次序通过每个结点指针有机地连接来体现。

(4) 插入和删除不需要大量移动表中元素。

要在链表中插入一个新结点怎样实现呢?设有线性表($a_1,a_2,\cdots,a_{i-1},a_i,\cdots,a_n$),采用单链表存储结构,头指针为 head,要求在数据元素 a_i的结点之前插入一个数据元素为 data 的新结点。插入前单链表的逻辑状态如图 8-5 所示。

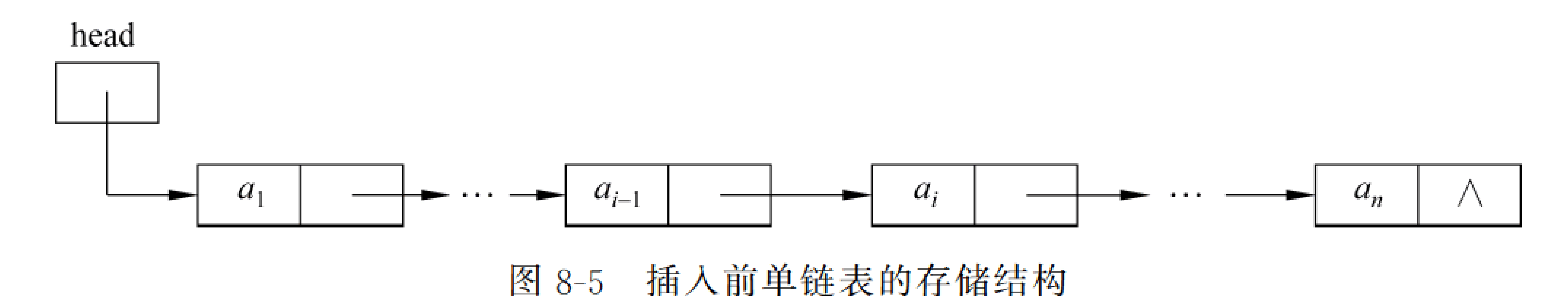


图 8-5　插入前单链表的存储结构

插入新结点后单链表的逻辑状态如图 8-6 所示。

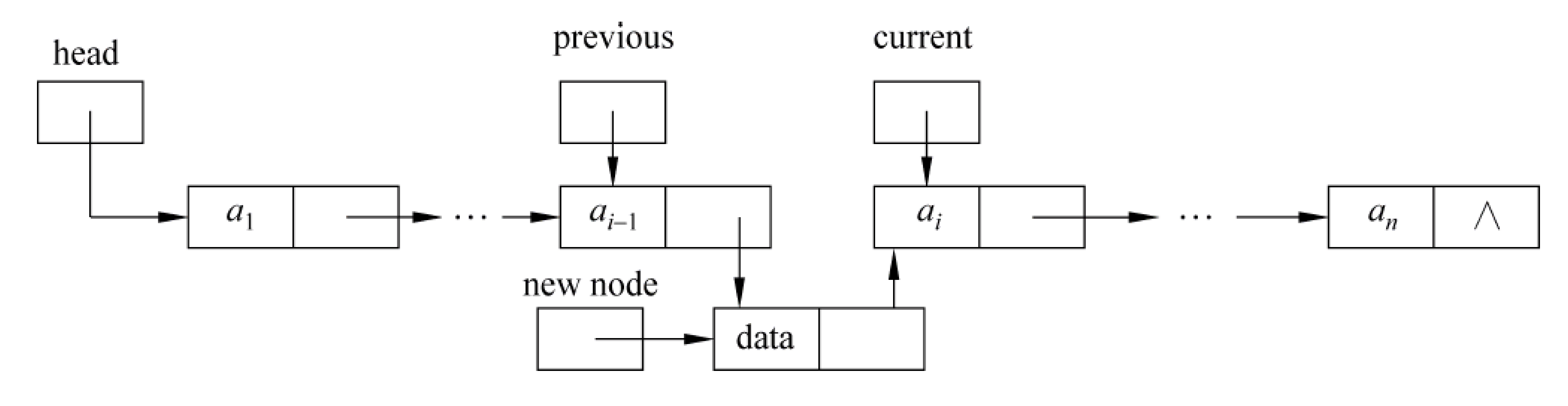


图 8-6　插入新结点后单链表的存储结构

若已知 previous 指向 a_i的前趋 a_{i-1}结点,newnode 指向新结点,只要执行以下两步操作即可完成插入新结点:

(1) 令新结点指针域指向 a_i结点(newnode－>next＝previous－>next)。

(2) 令 a_{i-1}结点的指针域指向新结点(previous－>next＝newnode)。

这就使得单链表成为如图 8-6 所示的插入后的逻辑状态。

由此可见,插入操作执行之前,首先要得到单链表中插入位置的前一个结点的指针(存储位置)。插入算法的主要步骤和程序描述如下:

步骤 1:判定插入位置是否正确,若正确继续下一步,否则结束算法。

步骤 2:申请一个新结点,判定内存有无空间(即表满否)。

步骤 3:元素放入数据域,NULL 放入指针域。

步骤 4:判定是否插入在表头,若是表头,则修改 head 和新结点指针。

步骤 5:寻找插入位置,指向 a_{i-1} 结点。

步骤 6：修改 a_{i-1} 结点的指针域和新结点的指针域。

删除操作和插入操作一样，首先要搜索单链表以找到指定删除结点的前趋结点(假设为 previous)，然后只要将待删除结点的指针域内容赋予 previous 所指向的结点的指针域就可以了。

已知单链表的头指针为 head，删除前单链表的逻辑状态如图 8-7 所示。

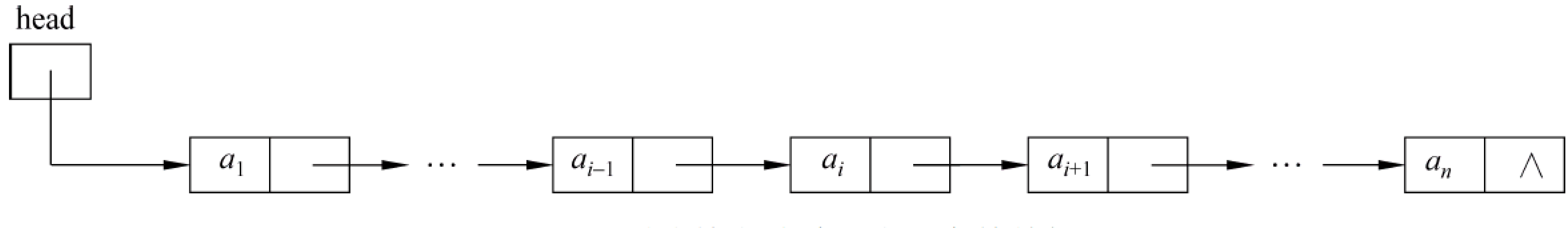


图 8-7　删除结点前单链表的存储结构

删除结点之后，单链表的逻辑状态如图 8-8 所示。

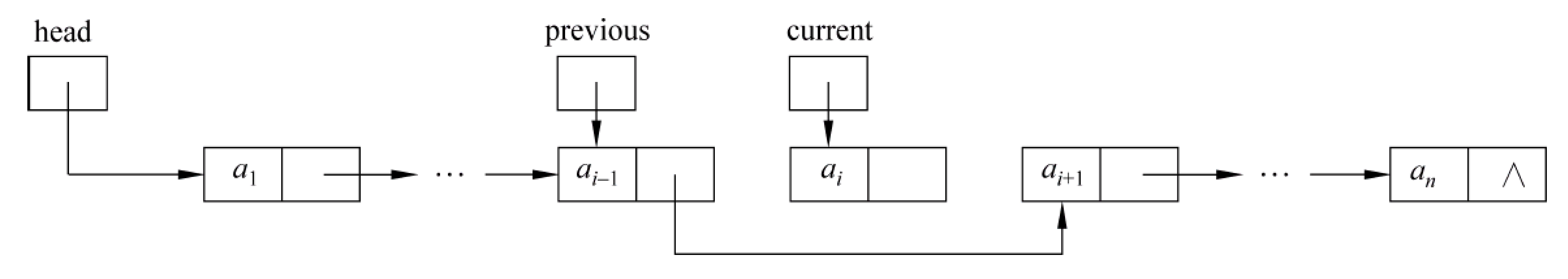


图 8-8　删除结点后单链表的存储结构

若已知 previous 指向 a_i 的前趋 a_{i-1} 结点，只要执行以下两步操作即可完成删除结点：

(1) 令 a_{i-1} 结点指针域指向 a_{i+1} 结点(previous－＞next＝current－＞next)。

(2) 释放 a_i 结点所占存储空间(delete current)。

这就使得单链表成为如图 8-8 所示的删除结点后的逻辑状态。

删除算法的主要步骤如下：

步骤 1：判定是否为空表，若为空表，删除错误，结束算法，否则继续下一步。

步骤 2：判定删除位置是否正确，若正确继续下一步，否则结束算法。

步骤 3：寻找删除位置，指向 a_{i-1} 结点。

步骤 4：若删除表头元素，修改 head 指针。

步骤 5：若删除非表头元素，修改 a_{i-1} 结点指针域。

步骤 6：释放被删结点的存储空间。

现在可以编写测试程序，测试插入函数和删除函数的正确性。参照前面测试顺序表的思路，分别对链表头插入和删除、链表尾插入和删除、链表的中间位置插入和删除、错误位置插入和删除进行验证，对不同类型的数据元素(如浮点型、字符串类)的链表也进行验证。

【例 8-3】　一个整数单链表的实现。

程序：

```
#include <stdlib.h>
#include <stdio.h>
//假设使用的是整数链表
```

```
typedef int DataType;
//单链表的结点定义
typedef struct NODE Node;
typedef struct NODE
{
    DataType  data;        //数据域
    Node*  next;           //指针域
}Node;
//头指针
typedef Node* Head;
//链表的方法声明
int GetLinkListLength(Head head);                          //求表长
void DestroyLinkList(Head head);                           //销毁链表
int GetElement(Head head, int n, DataType* data);          //获取线性表中第n个元素
int FindElement(Head head, DataType data);                 //查找data第一次出现的位置
int GetPriorElement(Head head, int n, DataType* data);     //获取第i个元素的前趋
int GetNextElement(Head head, int n, DataType* data);      //获取第i个元素的后继
int InsertToList(Head* head, int pos, DataType data);      //将data插入到pos处
int DeleteFromList(Head head, int pos);                    //删除线性表上位置为pos的元素
void PrintList(Head head);                                 //输出线性表
int InsertRear(Head* head, DataType data);                 //从表尾插入元素
int InsertHead(Head* head, DataType data);                 //从表头插入元素
//主函数
int main()
{
    //创建一个空的线性表
    Head head = NULL;
    //以下是对线性表的测试
    //插入5个元素并显示
    for (int i = 0; i < 5; ++i)
        InsertToList(&head, i, i + 1);
    //输出线性表
    PrintList(head);
    //在位置2插入99并显示
    printf("插入99后的表长:%d\n",InsertToList(&head, 2, 99));
    printf("插入99后的线性表\n");
    PrintList(head);
    //删除第4个元素
    printf("删除第4个元素后的表长:%d\n",DeleteFromList(head, 4));
    printf("删除第4个元素后的线性表\n");
    PrintList(head);
    //显示第3个元素的前趋
    DataType data;
    if (GetPriorElement(head, 3, &data) > -1);
```

```
    printf("第 3个元素的前趋是%d\n", data);
    DestroyLinkList(head);
    return 0;
}
//链表的方法实现
/* *
 * @brief  求表长
 * @param head  链表的头指针
 * @return  链表的长度
 * /
int GetLinkListLength(Head head)
{
    if (head == NULL)
        return 0;
    int i = 1;
    Node* pNode = head;
    while (pNode->next)                              //!= NULL
    {
        i++;
        pNode = pNode->next;                         //下一结点
    }
    return i;
}
/* *
 * @brief  销毁链表
 * @param head  链表的头指针
 * /
void DestroyLinkList(Head head)
{
    //从头开始,依次释放每一个结点
    Node* pNode;
    while (head)
    {
        pNode = head;
        head = head->next;                           //指向下一个结点
        free(pNode);                                 //释放当前结点
    }
}
/* *
 * @brief  获取线性表中第 n个元素,第一个元素的位置为 0
 * @param head  链表的头指针
 * @param n  要获取的元素位置
 * @param data  获取的数据存放在此
 * @return  获取成功返回 1,失败则返回 0
```

```
 * /
int GetElement(Head head, int n, DataType * data)
{
    if (n< 0 || n> GetLinkListLength(head) - 1)
        return 0;
    for (int i = 0; i < n; ++ i)
        head = head-> next;                          //移动到位置 n
    * data = head-> data;
    return 1;
}
/* *
 * @brief  查找 data 第一次出现的位置
 * @param head  指向线性表的头指针
 * @param data  要查找的元素
 * @return  找到则返回该位置,未找到返回-1
 * /
int FindElement(Head head, DataType data)
{
    int i = 0;
    while (head)
    {
        if (head-> data == data)                     //找到
            return i;
        head = head-> next;                          //下一个结点
        i++ ;
    }
    return - 1;
}
/* *
 * @brief  获取第 n 个元素的前趋
 * @param head  指向线性表的头指针
 * @param n  n 的前趋
 * @param data  获取成功,取得元素存放在 data 中
 * @return  找到则返回前趋的位置(n- 1),未找到返回-1
 * /
int GetPriorElement(Head head, int n, DataType * data)
{
    if (n< 1 || n> GetLinkListLength(head) - 1)
        return - 1;
    for (int i = 0; i < n - 1; ++ i)
        head = head-> next;                          //移动到 n- 1
    * data = head-> data;
    return n - 1;
}
```

```
/**
 *@brief  获取第 n个元素的后继
 *@param head  指向线性表的头指针
 *@param n   n的后继
 *@param data  获取成功,取得元素存放在 data中
 *@return  找到则返回后继的位置(n+1),未找到返回-1
 */
int GetNextElement(Head head, int n, DataType* data)
{
    if (n<0 || n>GetLinkListLength(head) -2)
        return -1;
    for (int i =0; i <n+1 ; ++i)
        head =head->next;                                   //移动到 n
    *data =head->data;
    return n +1;
}
/**
 *@brief  将 data插入到 pos处
 *@param head  指向线性表的头指针
 *@param pos  插入的位置
 *@param data  要插入的数据
 *@return  插入成功返回新的表长,否则返回-1
 */
int InsertToList(Head* head, int pos, DataType data)
{
    Node* pNode = *head;
    int length =GetLinkListLength(pNode);                   //得到表长
    if (pos<0 || pos>length)
        return -1;                                          //插入的位置不对
    if (pos ==0)                                            //在表头插入
        return InsertHead(head, data);
    if (pos ==length -1)                                    //在表尾插入
        return InsertRear(head, data);
    //定位到 pos的前一个位置
    for (int i =0; i <pos-1; ++i)
        pNode =pNode->next;
    //生成一个新的结点
    Node* pNewNode = (Node*)malloc(sizeof(Node));
    if (pNewNode==NULL)
        return -1;                                          //分配内存失败
    pNewNode->data =data;                                   //存入要插入的数据
    //插入链表
    pNewNode->next =pNode->next;
    pNode->next =pNewNode;
```

```
    //返回新的链表长度
    return ++length;
}
/**
 * @brief  删除线性表上位置为 pos 的元素
 * @param head  指向线性表的头指针
 * @param pos  删除的位置
 * @return  插入成功返回新的表长,否则返回-1
 */
int DeleteFromList(Head head, int pos)
{
    Node* pNode = head;
    int length = GetLinkListLength(head) ;                       //得到表长
    if (pos< 0 || pos> length - 1)
        return -1;                                               //删除的位置不对
    //定位到 pos 位置
    for (int i = 0; i < pos- 1; ++i)
        pNode = pNode-> next;
    Node* pDeleteNode = pNode-> next;
    pNode-> next = pNode-> next-> next;
    free(pDeleteNode);
    return --length;
}
/**
 * @brief  输出线性表
 * @param head  指向线性表的头指针
 */
void PrintList(Head head)
{
    int n = 0;
    while (head)
    {
        printf("第%d项元素为%d\n", n, head-> data);
        head = head-> next;
        ++n;
    }
}
/**
 * @brief  从表尾插入元素
 * @param head  指向线性表的头指针
 * @param data  插入的数据
 * @return  插入成功返回新的表长,否则返回-1
 */
int InsertRear(Head* head, DataType data)
```

```
{
    //准备新数据
    Node* pNewNode = (Node*)malloc(sizeof(Node));
    if (pNewNode == NULL)                              //内存分配失败
        return -1;
    pNewNode->data = data;
    if (*head == NULL)                                 //如果是空表
    {
        *head = pNewNode;
        pNewNode->next = NULL;
        return 1;                                      //表长为 1
    }
    //找到表尾
    Node* pNode = *head;
    while (pNode->next)
        pNode = pNode->next;
    //插入到表尾
    pNode->next = pNewNode;
    pNewNode->next = NULL;
    //返回新的表长
    return GetLinkListLength(*head);
}
/**
 * @brief 从表头插入元素
 * @param head 指向线性表的头指针
 * @param data 插入的数据
 * @return 插入成功返回新的表长,否则返回-1
 */
int InsertHead(Head* head, DataType data)
{
    //准备新数据
    Node* pNewNode = (Node*)malloc(sizeof(Node));
    if (pNewNode == NULL)                              //内存分配失败
        return -1;
    pNewNode->data = data;
    //插入
    if (*head == NULL)                                 //如果是空表
        pNewNode->next = NULL;
    else
        pNewNode->next = (*head)->next;
    *head = pNewNode;
    //返回新的表长
    return GetLinkListLength(*head);
}
```

程序的运行结果：

```
第 0项元素为 1
第 1项元素为 2
第 2项元素为 3
第 3项元素为 4
第 4项元素为 5
插入 99后的表长：6
插入 99后的线性表
第 0项元素为 1
第 1项元素为 2
第 2项元素为 99
第 3项元素为 3
第 4项元素为 4
第 5项元素为 5
删除第 4个元素后的表长：5
删除第 4个元素后的线性表
第 0项元素为 1
第 1项元素为 2
第 2项元素为 99
第 3项元素为 3
第 4项元素为 5
第 3个元素的前趋是 99
```

8.3 栈和队列

栈和队列也是线性结构，线性表、栈和队列这 3 种数据结构的数据元素以及数据元素间的逻辑关系完全相同，差别是线性表的操作不受限制，而栈和队列的操作受到限制。栈的操作只能在表的一端进行，队列的插入操作在表的一端进行而其他操作在表的另一端进行，所以，把栈和队列称为操作受限的线性表。

8.3.1 栈

栈是只能在一端插入和删除的特殊线性表。它按照后进先出的原则存储数据，先进入的数据被压入栈底，最后进入的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个进入的数据被第一个读出来）。

栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶（top），另一端为栈底（bottom）；栈底固定，而栈顶浮动；栈中元素个数为零时称为空栈。插入一般称为进栈（push），删除则称为出栈（pop）。栈也称为先进后出表。

图 8-9 是一个栈的示意图。

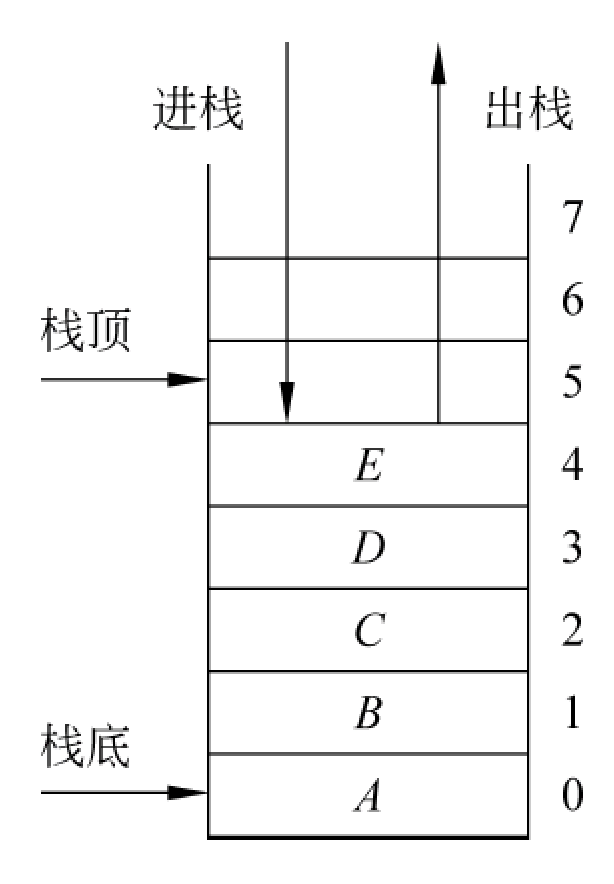


图 8-9　栈的示意图

栈顶始终指向栈中最后一个进入的元素之上的空位置。在图 8-9 中，栈中共有 5 个元素，入栈的次序依次是 *ABCDE*。栈底始终等于 0，而栈顶等于 5。图 8-10 则描述了最后两个元素出栈，*F* 进栈的情形。

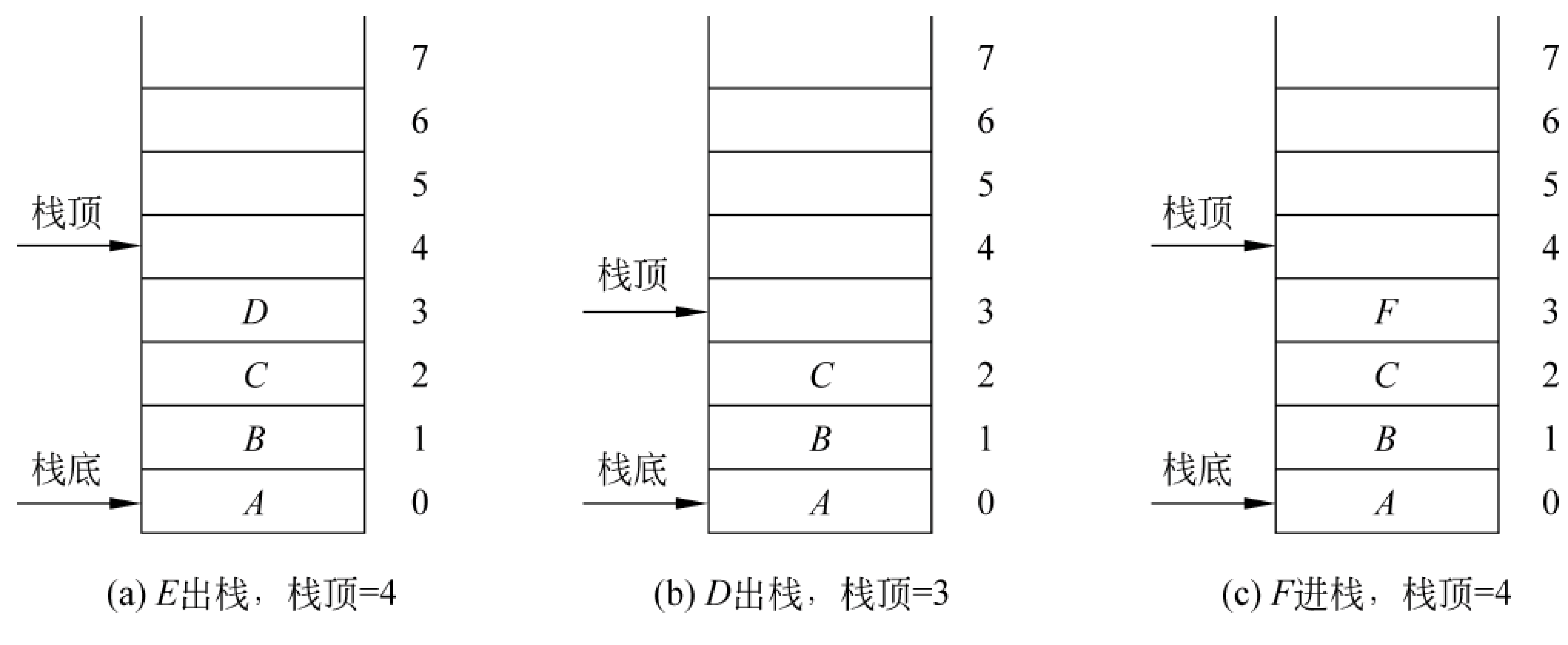


图 8-10　*E*、*D* 出栈，*F* 进栈

当栈中没有元素的时候称为空栈，空栈的条件是栈顶＝栈底。栈的大小一般是预先定义好的，当栈顶＝栈的大小时称为栈满。很显然，当栈为空的时候不能进行出栈操作，而当栈满的时候不能进行入栈操作。一般，对栈有如下几个操作：

- 求栈的长度：GetLength，返回栈中数据元素的个数。
- 判断栈是否为空：IsEmpty，如果栈为空返回 true，否则返回 false。
- 清空栈：Clear，使栈为空。
- 入栈操作：Push，将新的数据元素添加到栈顶，栈发生变化。
- 出栈操作：Pop，将栈顶元素从栈中取出，栈发生变化。
- 取栈顶元素：GetTop，返回栈顶元素的值，栈不发生变化。

同样，栈在计算机中的存储结构也有顺序存储和链式存储两种。显然顺序存储结构可以用数组来实现。

【例 8-4】　用数组实现栈。

分析：用一个指定大小的数组来存储栈的内容。在此，假设栈中存储的是字符串。

变量 top 保存栈顶的数组下标，变量 bottom 为栈底，始终为 0。

程序：

```
#include <stdlib.h>
#include<stdio.h>
//定义一个字符栈
typedef char DataType;
//定义栈
typedef struct STACK
{
    DataType* stackArray;
    int top;
    int bottom;
}Stack;
//声明栈的函数
Stack* CreateStack(int length);              //创建一个新的栈
void ClearStack(Stack* stack);               //清空栈
void DestroyStack(Stack* stack);             //销毁栈
DataType Pop(Stack* stack);                  //弹栈
void Push(Stack* stack, DataType data);      //压栈
int GetLength(Stack* stack);                 //得到栈的大小
DataType GetSatckPeek(Stack* stack);         //取得栈顶元素
//测试主函数
int main()
{
//定义栈的大小
const int MAXLENGTH=100;
//创建一个栈
Stack* stack=CreateStack(MAXLENGTH);
//输入 10个字符并入栈
for (int i=0; i<10; i++)
{
    char ch;
    scanf("%c", &ch);
    //入栈
    Push(stack, ch);
}
//弹栈并输出直到栈空
while (GetLength(stack)>0)
    printf("%c", Pop(stack));
DestroyStack(stack);
return 0;
}
//栈函数的实现
```

```
/**
  * @brief  创建一个新的栈
  * @param length  栈的大小
  * @return  指向栈的指针,如创建失败,返回 NULL
  */
Stack* CreateStack(int length)
{
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    if (stack)
    {
        //分配栈空间
        stack->stackArray = (DataType*)malloc(length* sizeof(DataType));
        if (stack->stackArray == NULL)
            return NULL;
        //置为空栈
        stack->bottom = 0;
        stack->top = 0;
    }
    return stack;
}
/**
 * @brief  清空栈
 * @param stack  指向栈的指针
 */
void ClearStack(Stack* stack)
{
    stack->bottom = 0;
    stack->top = 0;
}
/**
 * @brief  销毁栈
 * @param stack  指向栈的指针
 */
void DestroyStack(Stack* stack)
{
    free(stack->stackArray);
    free(stack);
}
/**
 * @brief  弹栈
 * @param stack  指向栈的指针
 * @return  弹出的栈顶元素,如果弹栈失败,返回 0
 */
DataType Pop(Stack* stack)
```

```
{
    if (stack->top > stack->bottom)
    {
        //如果栈不空,则出栈
        stack->top -= 1;
        return stack->stackArray[stack->top];
    }
    else
    {
        //栈已经空了,出栈失败
        return 0;
    }
}
/**
 * @brief  压栈
 * @param stack  指向栈的指针
 * @param data  要入栈的元素
 */
void Push(Stack* stack, DataType data)
{
    //此处没有处理栈满的情况,因为无法取得栈的最大容量
    stack->stackArray[stack->top] = data;
    stack->top++;
}
/**
 * @brief  得到栈的大小
 * @param stack  指向栈的指针
 * @return  栈大小
 */
int GetLength(Stack* stack)
{
    return stack->top - stack->bottom;
}
/**
 * @brief  取得栈顶元素,但是不出栈
 * @param stack  指向栈的指针
 * @return  栈顶元素,失败返回 0
 */
DataType GetSatckPeek(Stack* stack)
{
    return stack->top > stack->bottom ? stack->stackArray[stack->top - 1] : 0;
}
```

程序的运行结果如下:

0123456789
9876543210

8.3.2 队列

和栈相类似，队列也是一种特殊的线性表，它只允许在表的前端(front)进行删除操作，而在表的后端(rear)进行插入操作。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。在队列这种数据结构中，最先插入的元素将是最先被删除的元素；反之，最后插入的元素将是最后被删除的元素，因此队列又称为“先进先出”(First In First Out，FIFO)的线性表。

队列可以用数组来存储，数组的上界即是队列所容许的最大容量。在队列的运算中需设两个索引下标：front 为队头，存放实际队头元素的前一个位置；rear 为队尾，存放实际队尾元素所在的位置。一般情况下，两个索引的初值设为 0，这时队列为空，没有元素。图 8-11 是一个队列的示意图。

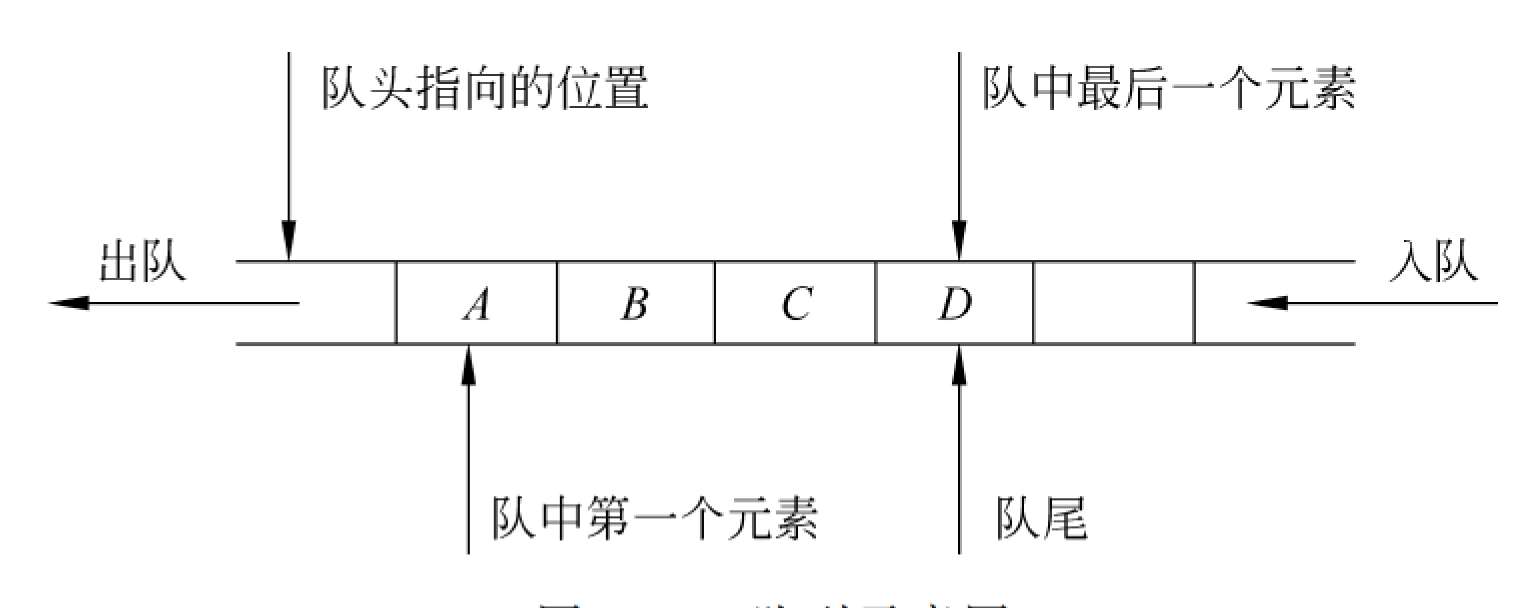


图 8-11 队列示意图

图 8-11 中，元素只能从队尾进入队列，只能从队头出队。也就是说要得到第 3 个元素 *C*，必须 *A* 和 *B* 先出队才可以。

当队头和队尾相等时，表示队列是空的，队尾到达了数组的上界，则队是满的。图 8-12 是一个队列变化的示意图。

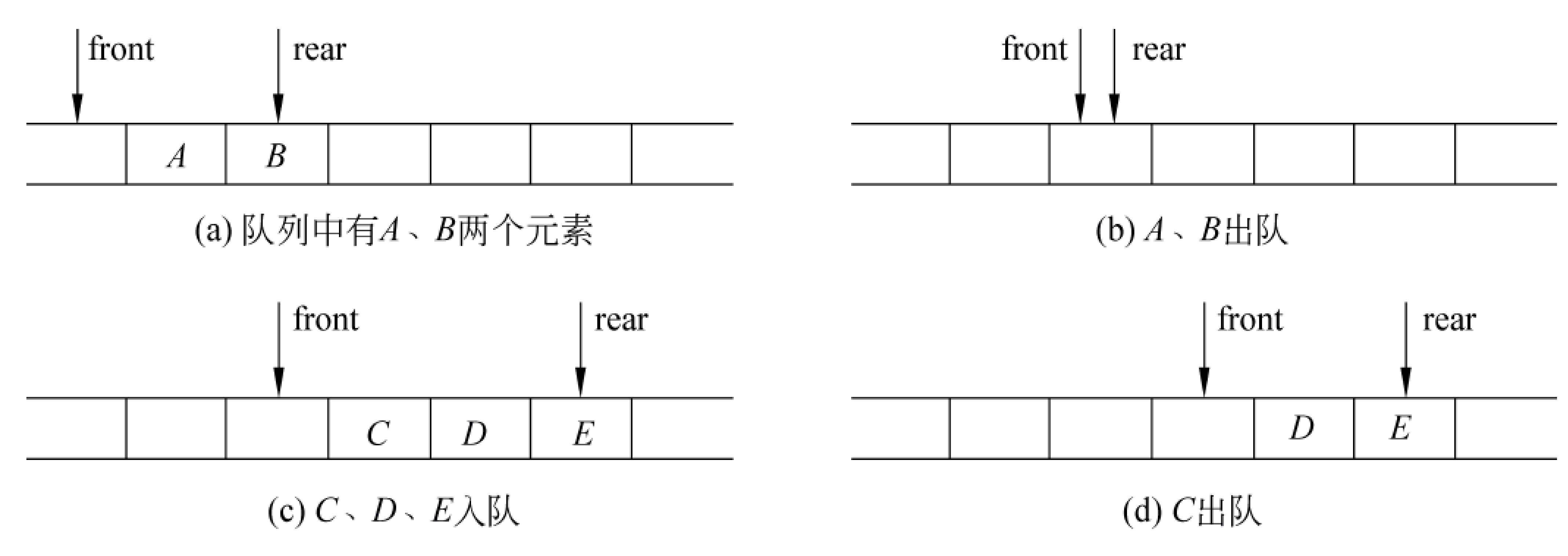


图 8-12 队列变化过程示意图

图 8-12(a)是这个队列中有两个元素 *A*、*B* 的情形；图 8-12(b)表示当 *A* 和 *B* 出队后，队列为空，此时队头等于队尾；图 8-12(c)为队中依次进入了 3 个元素 *C*、*D* 和 *E*；图 8-12(d)

为 C 出队后队列的情形。

对于一个队列,常用的操作如下:

- 求队列的长度 GetLength,得到队列中数据元素的个数。
- 判断队列是否为空 IsEmpty,如果队列为空返回 true,否则返回 false。
- 清空队列 Clear,使队列为空。
- 入队 EnQueue,将新数据元素添加到队尾,队列发生变化。
- 出队 DlQueue,将队头元素从队列中取出,队列发生变化。
- 取队头元素 GetFront,返回队头元素的值,队列不发生变化。

仔细观察图 8-12 的过程,会发现随着元素的出队和入队,队头和队尾均会不断地向后移动。当队尾移动到整个队列存储空间的最后一个位置时,如果还有元素要入队,则会发生溢出。因为队尾已经移到最后,没法再向后移动了。但实际上,队列中还是有空间的,因为有元素出队,也就是说,队头之前的空间是可以再用来存储数据的。如何利用空间呢?最直观的方法是将队列整个向前移动,但这样做效率并不高。一个较好的办法是将队列的头尾相连形成一个圆圈,这就是所谓的循环队列。循环队列的示意图如图 8-13 所示。当队尾和队头重叠时,队列为空还是满呢?我们约定,当队头和队尾相等时,队空。当队尾加 1 后等于队头时,队满。这样虽然浪费了一个存储空间(为什么?)但可以较为容易地区别队空和队满的情形。

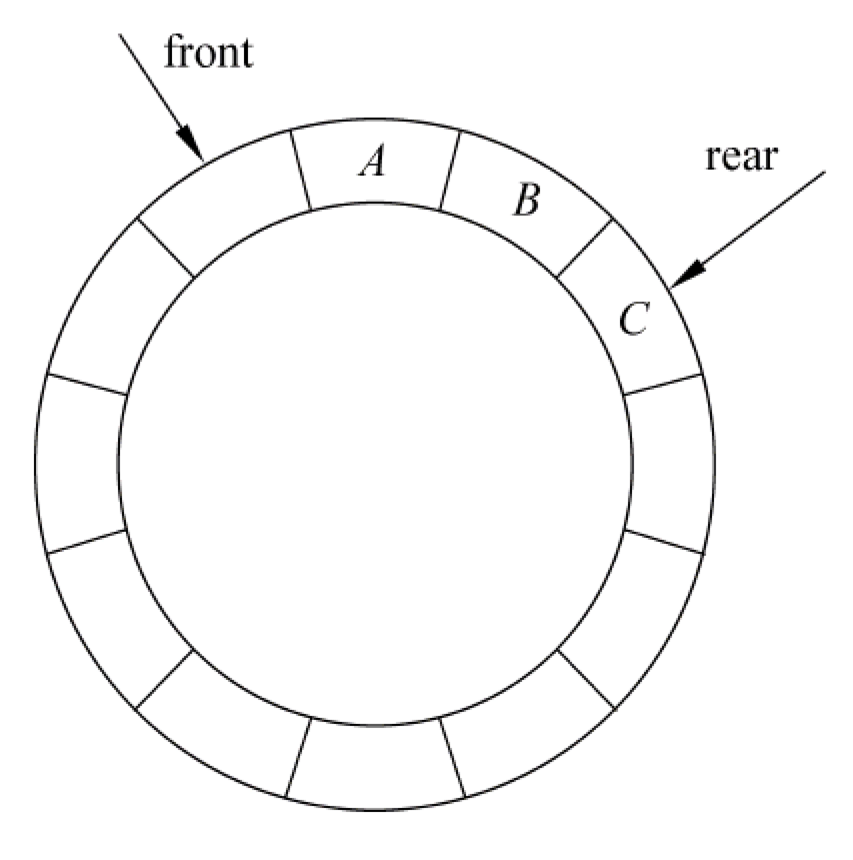


图 8-13　循环队列示意图

在循环队列中,空间可以反复利用。

【例 8-5】 一个队列的实现。

分析:使用固定大小的数组,实现队列的简单操作。

程序:

```
#include <stdio.h>
#include<stdlib.h>
typedef char DataType;              //假设是字符队列
//定义队列结构
typedef struct QUEUE
{
    DataType * queArray;
    int front;                      //队头
    int rear;                       //队尾
}Queue;

//声明队列的方法
Queue * CreateQueue(int length);    //创建一个队列
void DestroyQueue(Queue * queue);   //销毁队列
```

```
void ClearQueue(Queue* queue);                         //清空队列
int GetQueueLength(Queue* queue);                      //得到队列的长度
void EnQueue(Queue* queue, DataType data);             //入队
DataType DlQueue(Queue* queue);                        //出队
//主函数
int main()
{
    const int QueueMax = 100;                          //队列最大容量
    //创建队列
    Queue* queue = CreateQueue(QueueMax);
    if (queue == NULL)
        return 1;                                      //创建失败,程序退出
    //入队操作,3个元素进入队列
    EnQueue(queue, 'A');
    EnQueue(queue, 'B');
    EnQueue(queue, 'C');
    //一个元素出队,并显示
    printf("出队:%c\n", DlQueue(queue));
    //再入队 3个元素后打印队列
    EnQueue(queue, 'D');
    EnQueue(queue, 'E');
    EnQueue(queue, 'F');
    //所有元素依次出队,直到队空
    while (GetQueueLength(queue)>0)
        printf("出队:%c\n", DlQueue(queue));
    DestroyQueue(queue);
    return 0;
}

/**
 * @brief  创建一个队列
 * @param length  队列的容量
 * @return  指向队列的指针,失败返回 NULL
 */
Queue* CreateQueue(int length)
{
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    if (queue)                                         //不为空
    {
        //申请内存
        queue->queArray = (DataType*)malloc(length* sizeof(DataType));
        //失败则返回 NULL
        if (queue->queArray == NULL)
            return NULL;
```

```
        //清空队列
        queue->front = 0;
        queue->rear = 0;
    }
    return queue;
}
/**
 * @brief  销毁队列
 * @param queue  指向队列的指针
 */
void DestroyQueue(Queue* queue)
{
    free(queue->queArray);
    free(queue);
}
/**
 * @brief  清空队列
 * @param queue  指向队列的指针
 */
void ClearQueue(Queue* queue)
{
    queue->front = 0;
    queue->rear = 0;
}
/**
 * @brief  得到队列的长度
 * @param queue  指向队列的指针
 * @return  队列中的元素个数
 */
int GetQueueLength(Queue* queue)
{
    return queue->rear - queue->front;
}
/**
 * @brief  入队
 * @param queue  指向队列的指针
 * @param data  要入队的元素
 */
void EnQueue(Queue* queue, DataType data)
{
    //没有考虑队满的情况,这是因为没有记录队列的最大容量
    queue->queArray[++queue->rear] = data;
}
/**
```

```
 * @brief  出队
 * @param queue  指向队列的指针
 * @return  出队的元素值,如队空,返回 0
 * /
DataType DlQueue(Queue *  queue)
{
    return queue-> rear - queue -> front > 0 ?queue-> queArray[++ queue-> front] : 0;
}
```

程序的运行结果：

```
出队：A
出队：B
出队：C
出队：D
出队：E
出队：F
```

【例 8-6】 循环队列。

分析：使用固定大小的数组实现一个循环队列。这里的代码和例 8-5 基本是相同的，只是在入队、出队和打印等操作上注意队头和队尾对队列的总长度取余。此外还要注意，本例在队列的定义中增加了一项来记录队列的最大容量。

代码：

```
#include < stdio.h>
#include< stdlib.h>
typedef char DataType;                          //假设是字符队列
//定义队列结构
typedef struct QUEUE
{
    DataType *  queArray;
    int front;                                  //队头
    int rear;                                   //队尾
    int maxLength;                              //队的最大容量
}Queue;
//声明队列的方法
Queue *  CreateQueue(int length);               //创建一个队列
void DestroyQueue(Queue *  queue);              //销毁队列
void ClearQueue(Queue *  queue);                //清空队列
int GetQueueLength(Queue *  queue);             //得到队列的长度
void EnQueue(Queue *  queue, DataType data);    //入队
DataType DlQueue(Queue *  queue);               //出队
//主函数
int main()
{
```

```
    const int QueueMax = 100;                    //队列最大容量
    //创建队列
    Queue * queue = CreateQueue(QueueMax);
    if (queue == NULL)
        return 1;                                //创建失败,程序退出
    //入队操作,3个元素进入队列
    EnQueue(queue, 'A');
    EnQueue(queue, 'B');
    EnQueue(queue, 'C');
    //一个元素出队,并显示
    printf("出队:%c\n", DlQueue(queue));
    //再入队 3个元素后打印队列
    EnQueue(queue, 'D');
    EnQueue(queue, 'E');
    EnQueue(queue, 'F');
    //所有元素依次出队,直到队空
    while (GetQueueLength(queue)> 0)
    printf("出队:%c\n", DlQueue(queue));
    DestroyQueue(queue);
    return 0;
}
/**
 * @brief  创建一个队列
 * @param length  队列的容量
 * @return  指向队列的指针,失败返回 NULL
 */
Queue * CreateQueue(int length)
{
    Queue * queue = (Queue * )malloc(sizeof(Queue));
    if (queue)                                   //不为空
    {
        //申请内存
        queue->queArray = (DataType * )malloc(length * sizeof(DataType));
        //失败则返回 NULL
        if (queue->queArray == NULL)
            return NULL;
        //清空队列
        queue->front = 0;
        queue->rear = 0;
        queue->maxLength = length;
    }
    return queue;
}
/**
```

```
 * @brief  销毁队列
 * @param queue  指向队列的指针
 * /
void DestroyQueue(Queue * queue)
{
    free(queue->queArray);
    free(queue);
}
/* *
 * @brief  清空队列
 * @param queue  指向队列的指针
 * /
void ClearQueue(Queue * queue)
{
    queue->front = 0;
    queue->rear = 0;
}
/* *
 * @brief  得到队列的长度
 * @param queue  指向队列的指针
 * @return  队列中的元素个数
 * /
int GetQueueLength(Queue * queue)
{
    return queue->rear >= queue->front ?
        queue->rear - queue->front :
        queue->rear - queue->front + queue->maxLength;
}
/* *
 * @brief  入队
 * @param queue  指向队列的指针
 * @param data  要入队的元素
 * /
void EnQueue(Queue * queue, DataType data)
{
    if ((queue->rear + 1) % queue->maxLength == queue->front)
    {
        printf("队列已满,无法完成入队操作");
    }
    else
    {
        //队尾向后移动 1 位
        queue->rear = (queue->rear + 1) % queue->maxLength;
        //入队
```

```
            queue->queArray[queue->rear] = data;
        }
}
/**
  * @brief  出队
  * @param queue  指向队列的指针
  * @return  出队的元素值,如队空,返回 0
  */
DataType DlQueue(Queue * queue)
{
    if (GetQueueLength(queue) > 0)              //队不空
    {
        queue->front = (queue->front + 1) % queue->maxLength;
        return queue->queArray[queue->front];
    }
    else
    {
        return 0;                               //队是空的,返回 0
    }
}
```

8.4　图　和　树

前几节讲述了一些线性结构的数据,而图和树则是非线性的数据结构。同时,现实中的很多问题也是用线性数据结构无法描述的,需要借助非线性的数据结构来描述。

8.4.1　图的基本概念

1736 年,著名数学家欧拉(Euler)发表了著名的论文《柯尼斯堡七座桥》,首先使用图的方法解决了柯尼斯堡七桥问题,从而欧拉也被誉为图论之父。这个问题是基于一个现实生活中的事例:当时东普鲁士柯尼斯堡(Königsberg,今日俄罗斯加里宁格勒)市区跨普列戈利亚河(Pregel)两岸,河中心有两个小岛。小岛与河的两岸有 7 座桥连接。于是,7 座桥将 4 块陆地连接了起来,如图 8-14 所示。而城里的居民想在散步的时候从任何一块陆地出发,经过每座桥一次且仅经过一次,最后返回原来的出发点。当地的居民和游客做了不少尝试,却都没有成功,而欧拉最终解决了这个问题并断言这样的回路是不存在的。

欧拉在解决问题时,用 4 个结点来表示陆地 A、B、C 和 D,凡是陆地间有桥连接的,便在两点间连一条线,于是图 8-14 转换为图 8-15。

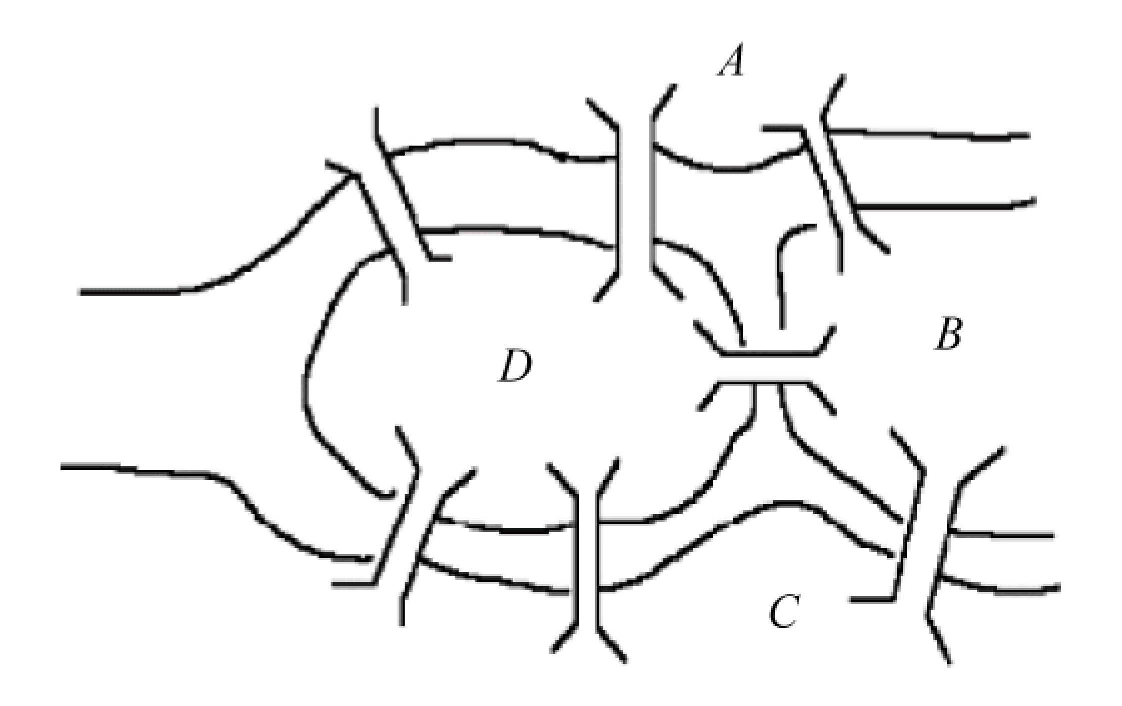


图 8-14 柯尼斯堡七桥问题示意图

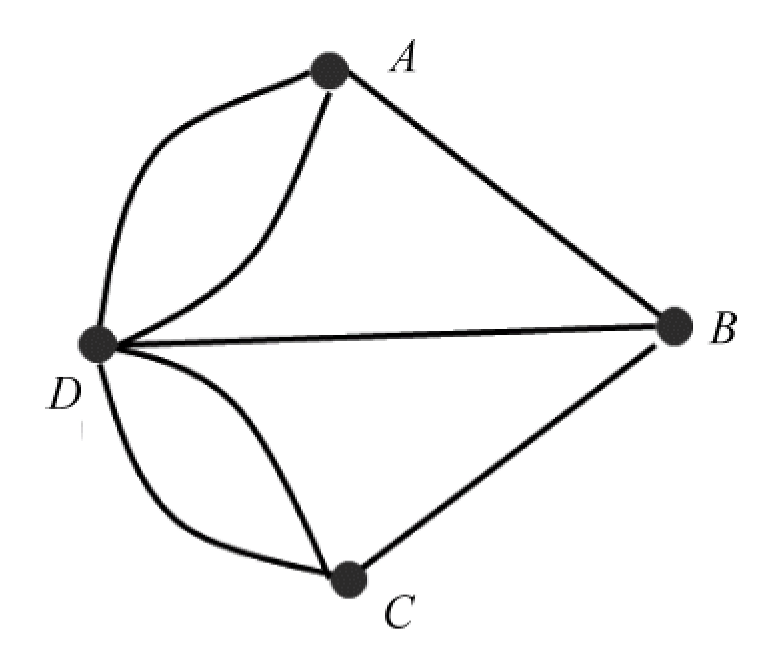


图 8-15 柯尼斯堡七桥问题抽象为图后的表示

此时，问题则转化为：从图 8-15 中的 A、B、C、D 任一点出发，通过每条边一次且仅一次后回到原出发点的回路是否存在？欧拉断言了这个回路是不存在的，理由是从图 8-15 中的任一点出发，为了能够回到原出发点，则要求与每个点关联的边数均为偶数。这样才能保证从一条边进入某点后可以从另外一条边出来。而图 8-14 中的 A、B、C、D 全部都与奇数边关联，因此回路是不存在的。

由上面的例子可以看到，所谓图(graph)是由结点或称顶点(vertex)和连接结点的边(edge)所构成的图形。使用 $V(G)$表示图 G 中所有结点的集合，$E(G)$表示图 G 中所有边的集合。则图 G 可记为$<V(G),E(G)>$或$<V,E>$。有 n 个顶点和 m 条边的图记为(n,m)图或称为 n 阶图。

【例 8-7】 有 4 个城市 v_1、v_2、v_3 和 v_4。v_1 和其他 3 个城市都有道路连接，v_2 和 v_3 之间有道路连接，画出图并用集合表示该图。

显然结点集合 $V=\{v_1,v_2,v_3,v_4\}$，边集合 $E=\{v_1$ 和 v_2 之间的边，v_1 和 v_3 之间的边，v_1 和 v_4 之间的边，v_2 和 v_3 之间的边$\}$。画出的图如图 8-16 所示。

更一般地，边可以用结点对来表示，或者说用结点 V 的向量积来表示：

$V=\{v_1,v_2,v_3,v_4\}$

$E=\{(v_1,v_2),(v_1,v_3),(v_1,v_4),(v_2,v_3)\}$

在图中，如果边不区分起点和终点，这样的边称为无向边。所有边都是无向边的图称为无向图，如图 8-16 就是一个无向图。反之，若边区分起点和终点，则为有向边，所有边都是有向边的图称为有向图。在图中，有向边使用带有箭头的线段表示，由起点指向终点。在集合中则用有序对$<v_1,v_2>$来表示，图 8-17 是一个示例。

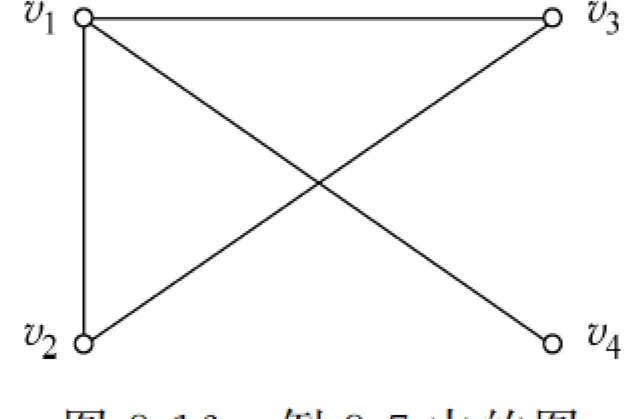


图 8-16 例 8-7 中的图

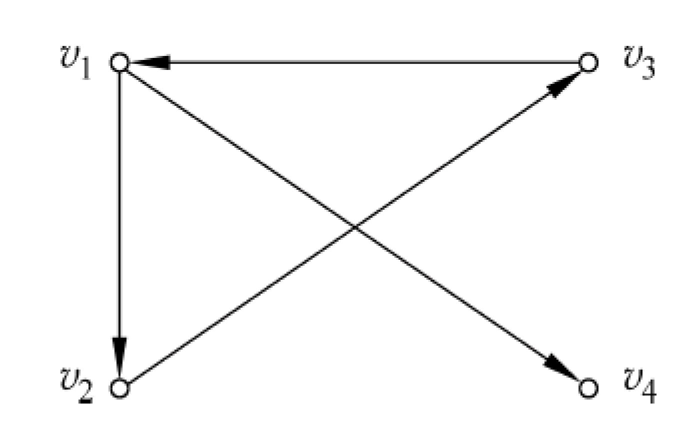


图 8-17 一个有向图的示例

在图 8-17 中：

$V=\{v_1,v_2,v_3,v_4\}$

$E=\{<v_1,v_2>,<v_1,v_4>,<v_3,v_1>,<v_2,v_3>\}$

结点的度则是指和结点关联的边的个数。如在图 8-17 中，v_1 的度是 3，v_2 和 v_3 的度是 2，v_4 的度是 1。对于有向图，则区分为出度和入度，由结点指向外的边的个数为出度，反之为入度。如图 8-17 中，v_1 的出度为 2，入度为 1，v_4 的出度为 0，入度为 1。

图在计算机中如何存储则是人们普遍关心的一个问题。简单的方法是将图用一个二维矩阵来表示，这样的矩阵通常称为邻接矩阵。在此不作系统讨论，仅以图 8-17 的存储为例来说明。

【例 8-8】 将简单有向图(图 8-17)以邻接矩阵的方式存储到计算机中。

要以邻接矩阵的方式存储，首先需要对结点指定一个次序。在此，就以结点的下标从小到大为序，排列为 v_1，v_2，v_3，v_4。然后使用一个 4×4 的矩阵来存储该图，矩阵中的元素只有两个取值：0 或者 1。对于两个结点 v_i 和 v_j，若 v_i 和 v_j 之间存在一条边，则对应的矩阵元素 $a_{ij}=1$，反之则为 0。图 8-17 中的有向图的矩阵如图 8-18 所示。

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array}\begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

图 8-18　存储的邻接矩阵

容易看出，矩阵中 1 的个数对应图中边的个数，而对角线的元素则全为 0。

8.4.2　带权图和最短路径

图的问题异常复杂，甚至是一门完整的学科——图论。在此不对图作系统的讨论。而为了使读者对图有进一步的认识，作为一个例子，简单介绍带权图及最短路径的算法，并以此结束对图的讨论。

在处理有关图的实际问题时，往往有值的存在，比如公里数、运费、城市、人口数以及电话部数等。一般这个值称为权值，在图中，将每条边都有一个非负实数对应的图称为带权图或赋权图。这个实数称为这条边的权。根据不同的实际情况，权数的含义可以各不相同。例如，可用权数代表两地之间的实际距离或行车时间，也可用权数代表某工序所需的加工时间等。如图 8-19 所示便是一个带权图。

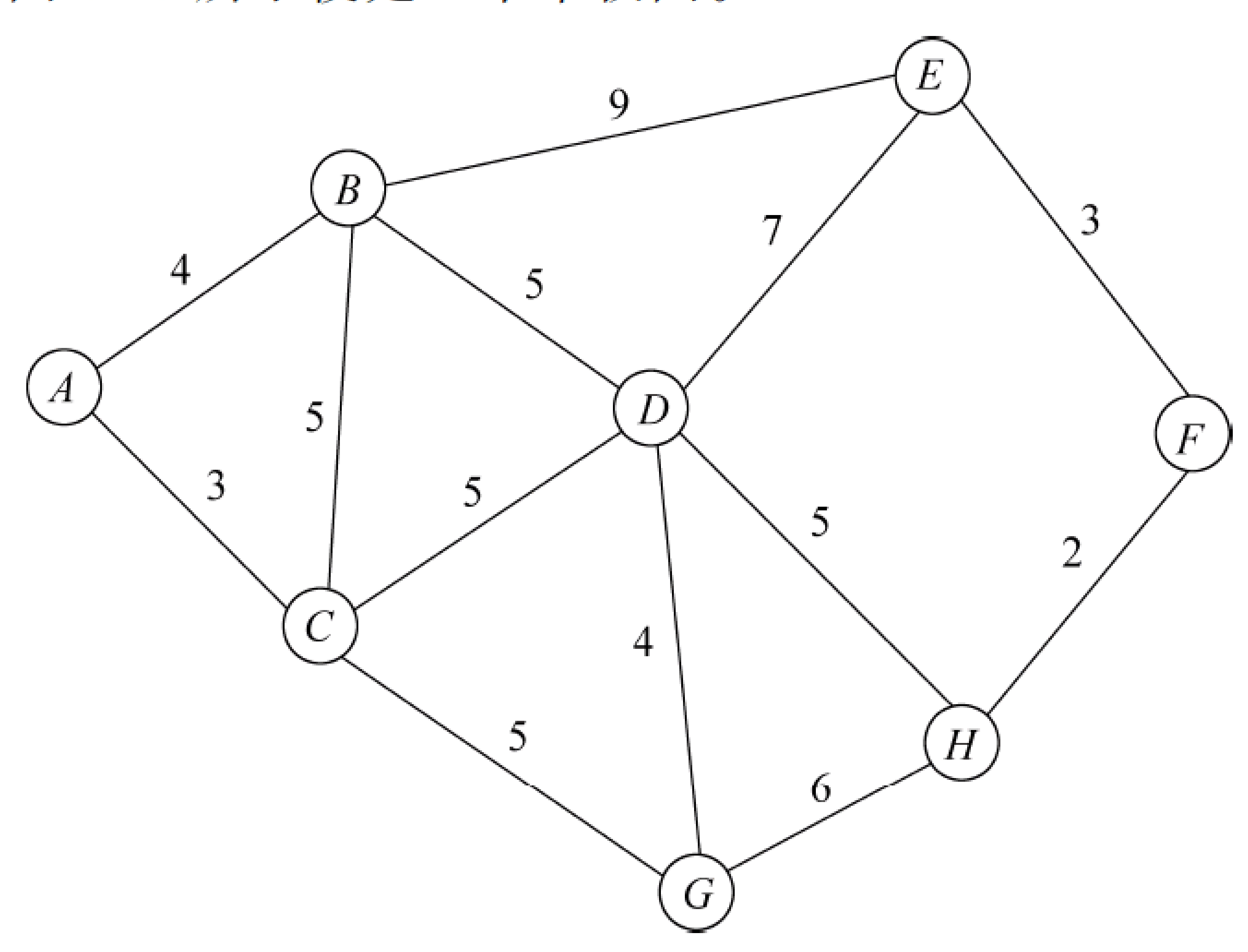


图 8-19　带权图

对图 8-19 所示的无向带权图求最短路径是一个经常遇到的问题。假设在图中的 A 到 G 点表示 8 个村庄，边表示村庄之间的道路。边上的权值表示距离。现在的问题是，从 A 到 F 最短的距离是多少？

求最短路径的算法是 E. W. Dijkstra 于 1959 年提出来的，这是至今公认的求最短路径的最好方法，称为 Dijkstra 算法。假定给定带权图 G，要求 G 中从 v_0 到 v 的最短路径，Dijkstra 算法的基本思想是：

将图 G 中结点集合 V 分成两部分：一部分称为具有 P 标号的集合，另一部分称为具有 T 标号的集合。所谓结点 a 的 P 标号是指从 v_0 到 a 的最短路径的路长；而结点 b 的 T 标号是指从 v_0 到 b 的某条路径的长度。Dijkstra 算法中首先将 v_0 取为 P 标号结点，其余的结点均为 T 标号结点，然后逐步地将具有 T 标号的结点改为 P 标号结点，当目的结点也被改为 P 标号时，则找到了从 v_0 到 v 的一条最短路径。下面通过一个例子给出实际的算法步骤。

【例 8-9】 计算图 8-19 所示的带权图中从 A 点到 F 点的最短路径。

(1) 首先，将起点 A 划归为 P 标号集合，其余的结点均为 T 结点。A 到 A 的距离为 0，所以 A 的 P 标号为 0。

(2) 更新 T 中结点到 A 的距离。如和 A 相邻(有边连接)，则距离就是边的权值。如和 A 没有直接的边连接，则距离是无穷大。

(3) 在 T 中找到一个值最小的结点，并将其划归到 P 集合。

(4) 根据新进入的 C 结点，更新与 C 相连的结点的值。新值等于 C 的 P 结点值加上到与其相连的结点的距离(边的权值)。更新的算法是：如果新值小于原有的值，则用新的值取代，否则保持原有值不变。

(5) 重复步骤(3)和(4)，直到目标点进入 P 集合。

图 8-20 到图 8-26 演示了上述过程。

图 8-20 中，A 到 B 的距离为 4，到 C 的距离为 3，到其余结点的距离为无穷大。由于 C 结点的值最小，因此 C 进入 P 集合(P 集合以方框表示，T 集合用圆圈表示)。

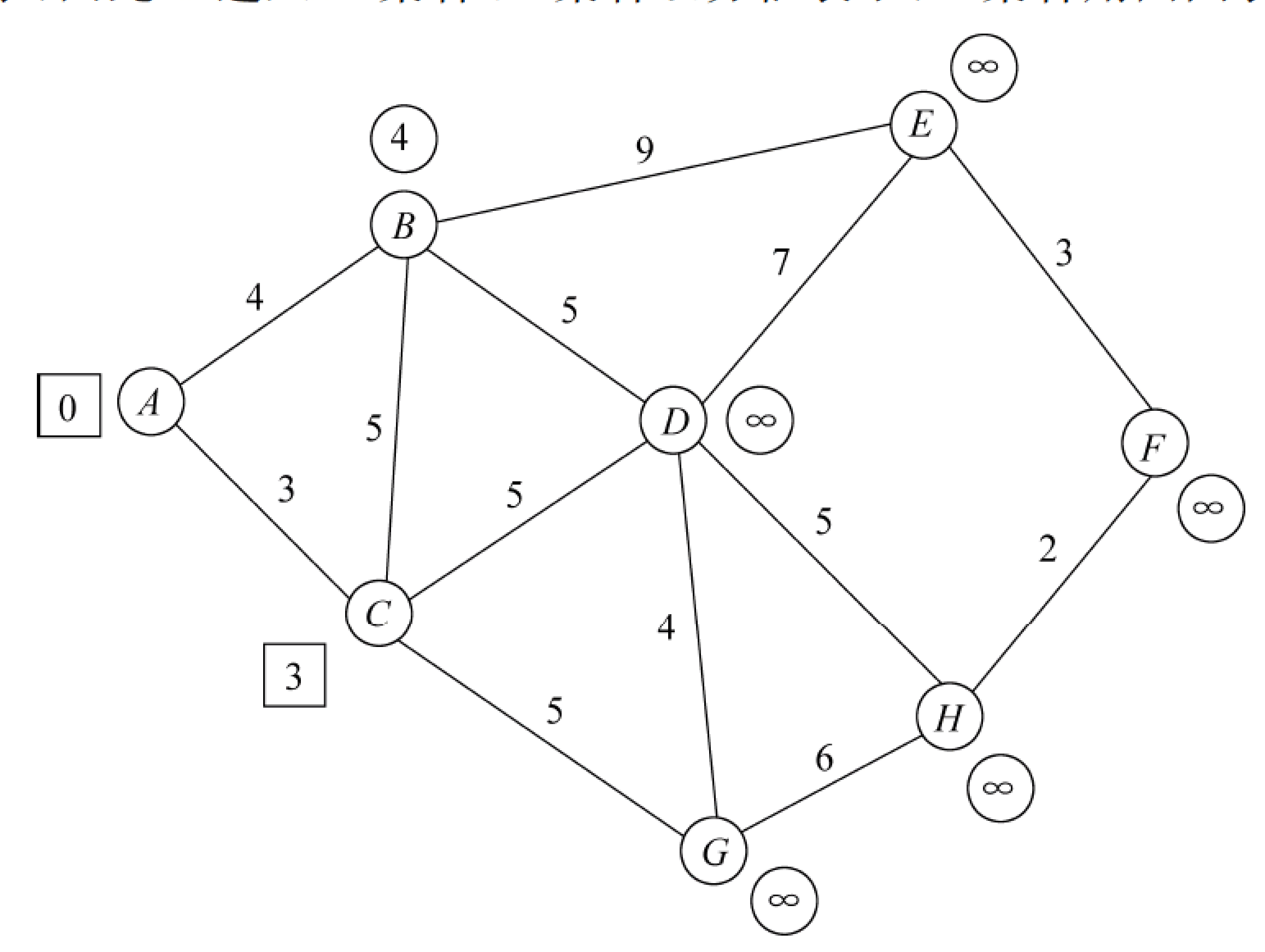


图 8-20 C 进入 P 集合

图 8-21 中，结点 C 进入 P 集合后，到 B 的距离为 3＋5＝8，大于 B 原来的 4，因此 B 的值不变。而到 D 和 G 的值均为 8，均小于原来的无穷大，因此用 8 取代原来的值。之后，在 T 中，B 的值为 4 最小，B 进入 P 集合。

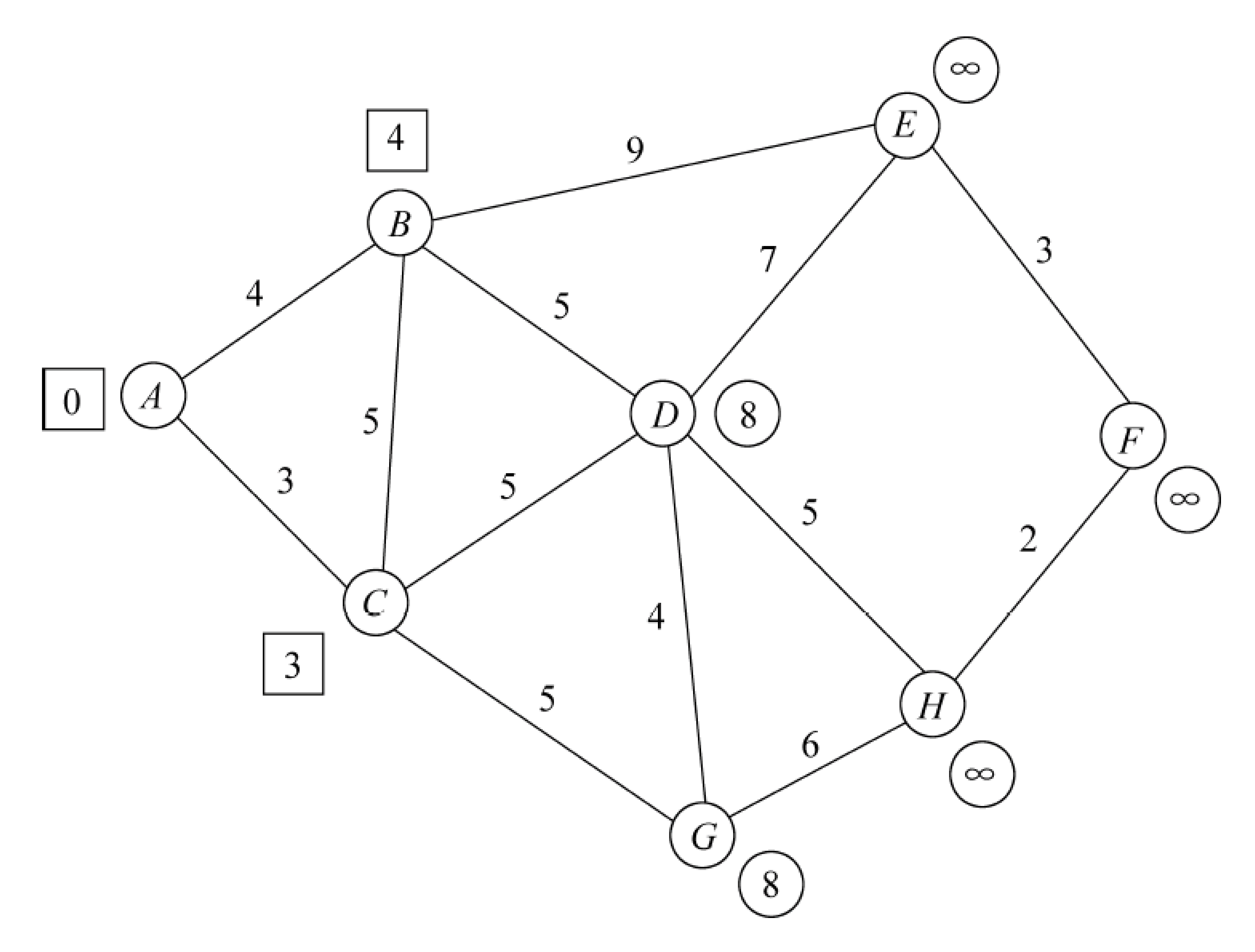


图 8-21　B 进入 P 集合

图 8-22 中，结点 B 进入后更新与 B 连接的 D 和 E 值。其中 D 的值不变，E 为 13。此时 D 和 G 均有最小值 8，任取一个结点进入 P，在此取的是 D。然后又更新了 H 的值。G 的原值小于 8＋4，因此保持不变。

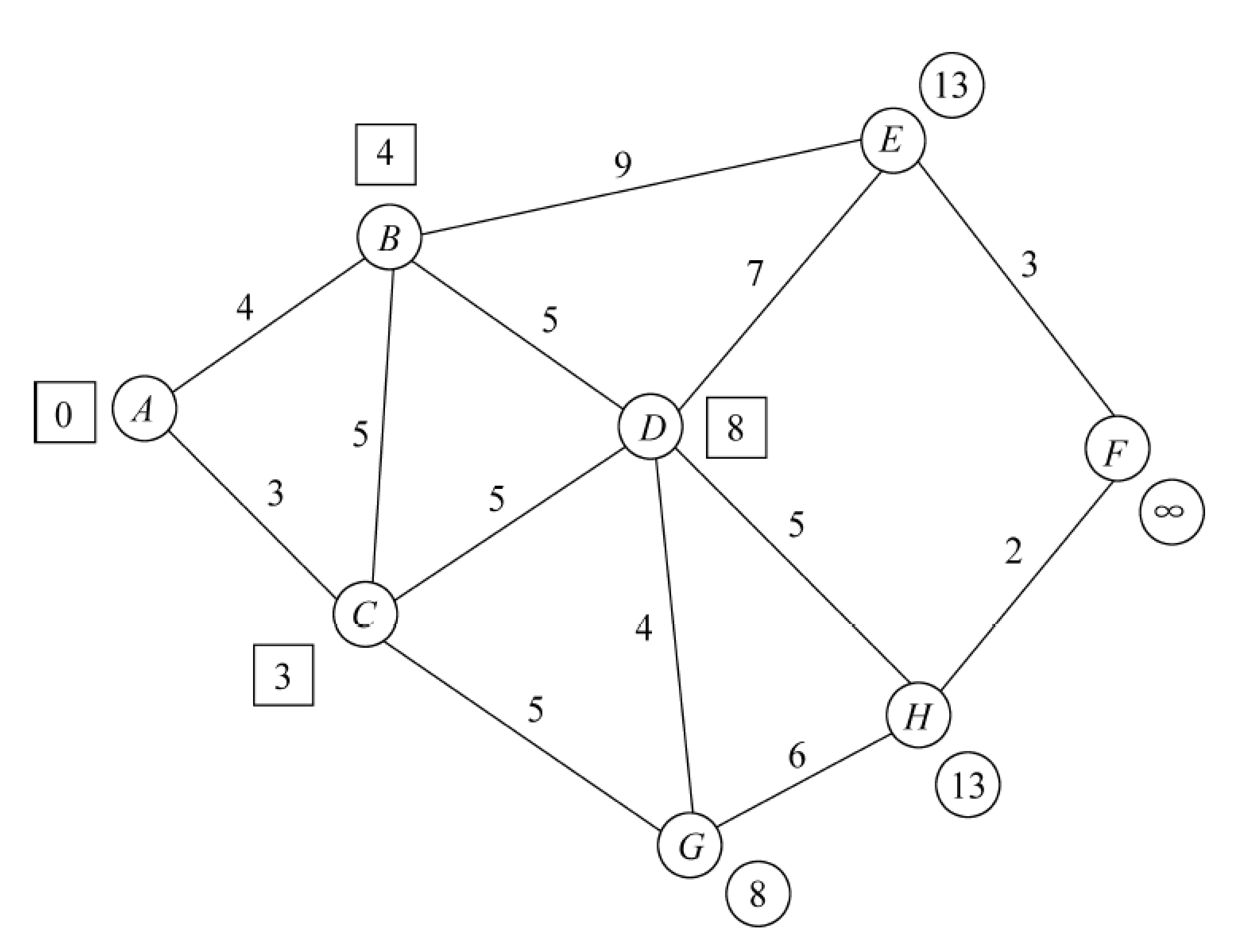


图 8-22　D 进入 P 集合

图 8-23 中，结点 G 的值最小，G 进入 P 集合。H 的值未变。

图 8-24 中，任选 E 进入 P 集合，F 值变为 16。

图 8-25 中，结点 H 进入 P 集合，F 的值变为 15。

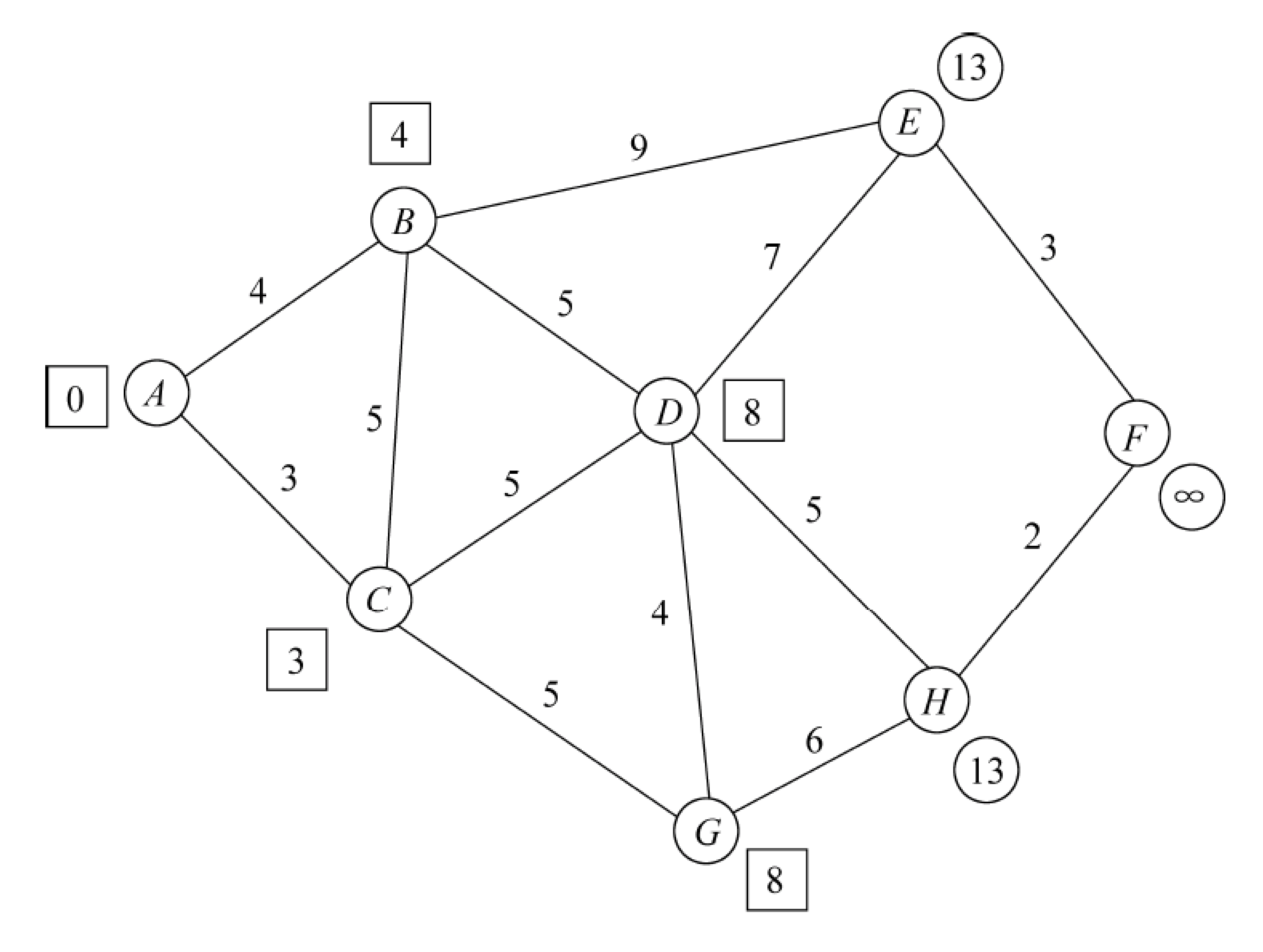


图 8-23　*G* 进入 *P* 集合

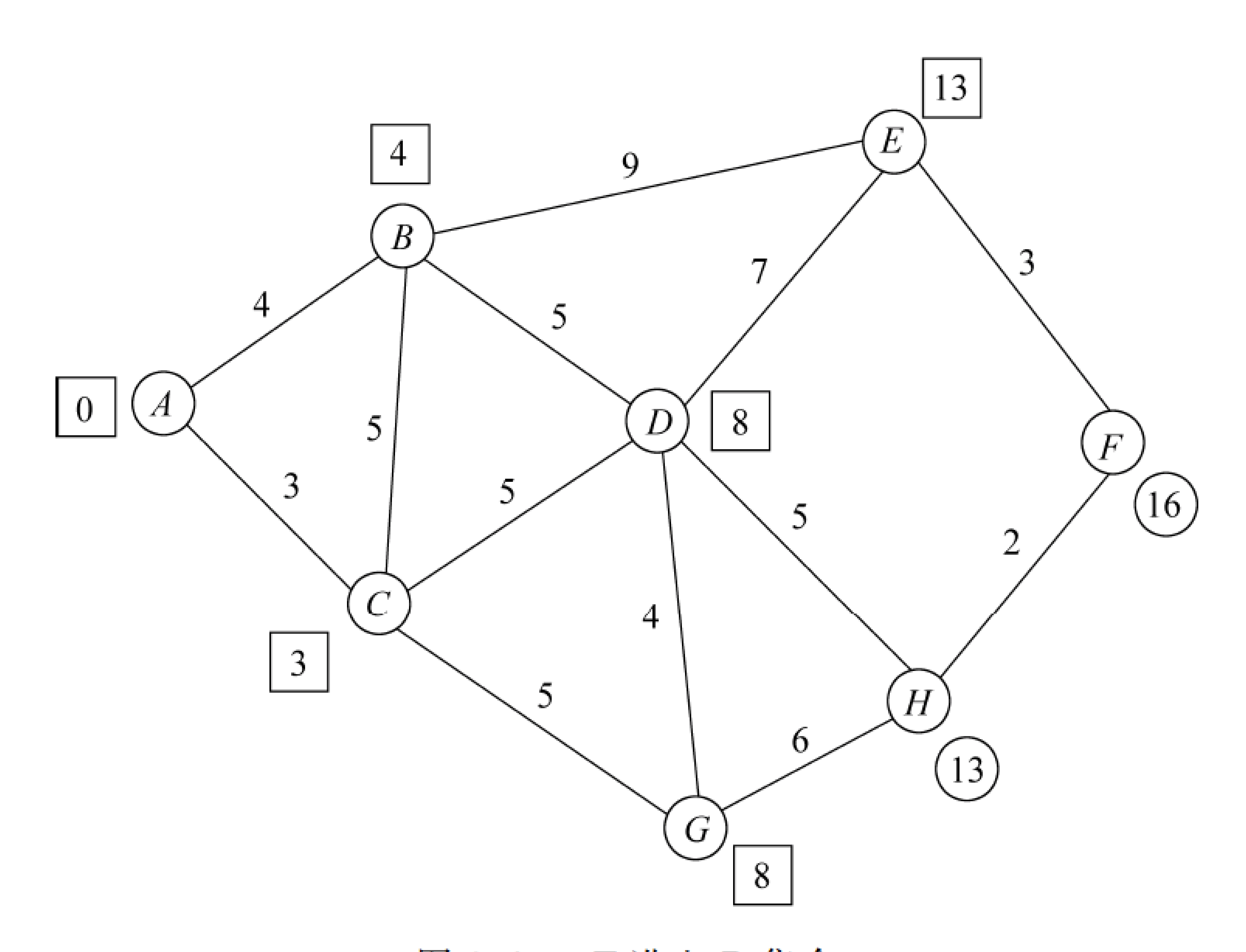


图 8-24　*E* 进入 *P* 集合

图 8-26 中，终点 *F* 进入 *P* 集合，运算结束。从 *A* 到 *F* 的最短距离为 15。而事实上，对于每一个 *P* 中的结点值，计算出了从 *A* 到该结点的最短距离，如到 *E* 的最短距离为 13。而找到最短路径的方法是用 *F* 点的 *P* 值减去边的权值，倒推回 *A* 点。如 *F* 的值为 15－2＝13，和 *H* 吻合，而不是 *E*(因为 15－3＝12，不等于 *E* 的 13)。

8.4.3　树的基本概念

树可以看作一个特殊的有向图。对于一个有向图，如果

(1) 存在一个特殊的结点 *r*，其入度等于 0。

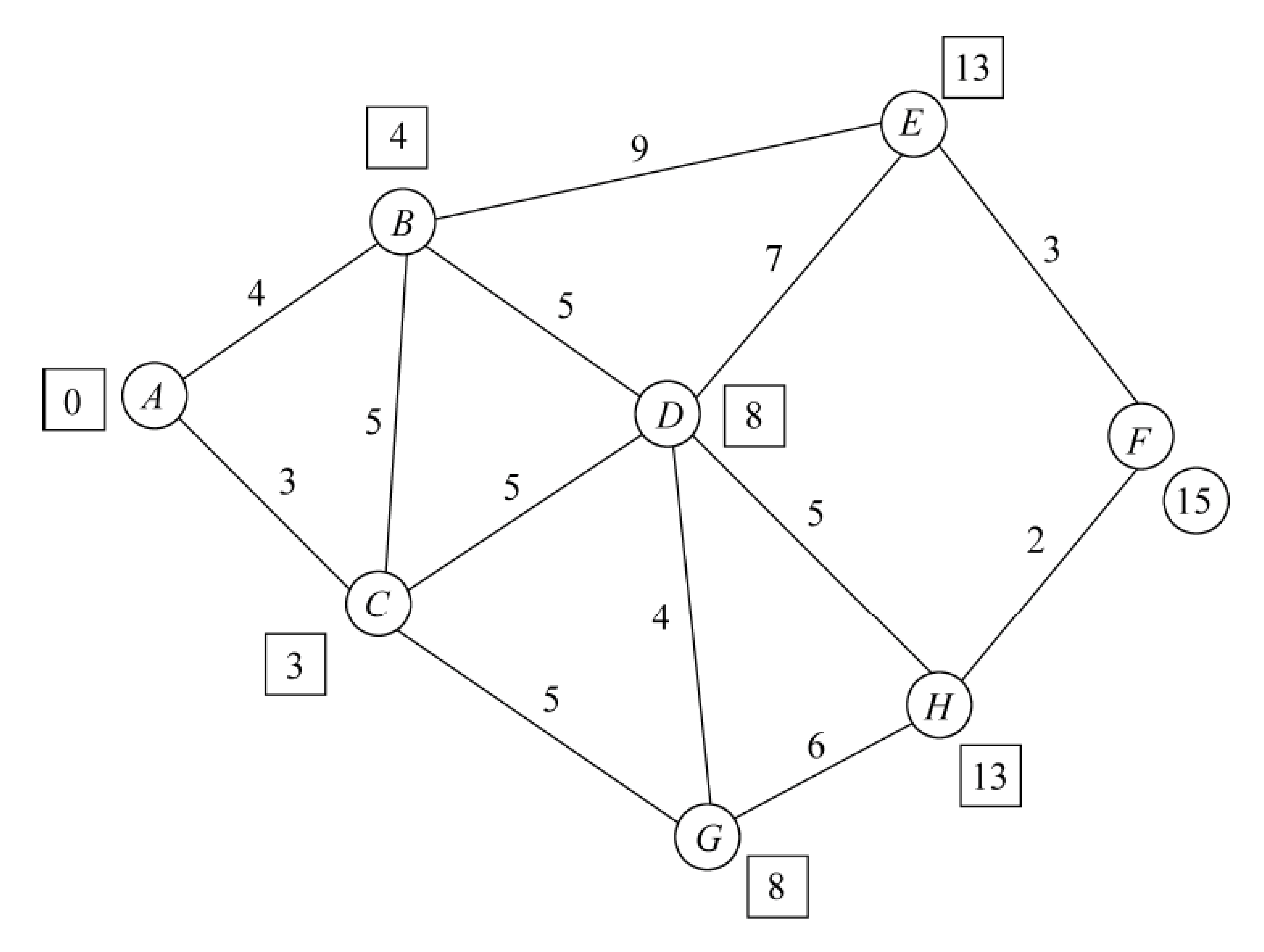


图 8-25　H 进入 P 集合

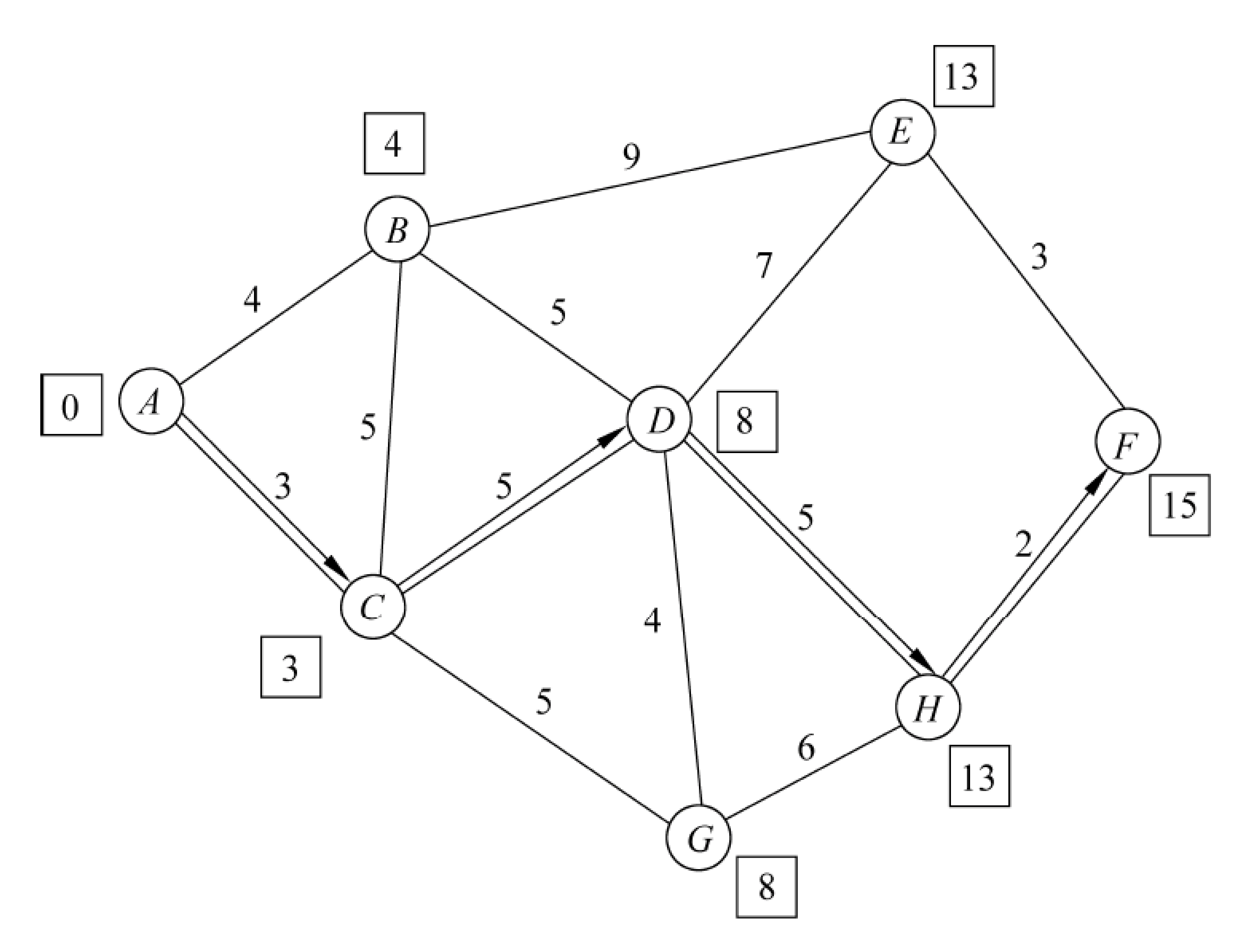


图 8-26　求得最短路径

(2) 除了 r 外的其他结点的入度均为 1。

(3) R 到图中其他结点均有路可达。

满足这样 3 个条件的图称为树。其中入度为 0 的结点称为根，出度为 0 的结点称为叶子结点。出度不为 0 的结点称为分枝结点，如图 8-27 所示。

在画树的时候，由于所有的箭头方向都是一致的，所以箭头常常省略，如图 8-28 所示。树是有层次的，指的是从根到该结点的距离。称距根最远的叶子的层数为树的高度。图 8-28 的树的高度为 3。同一层次之间的结点称为兄弟，上一层次的结点为父亲，下一层次的结点是儿子，如图 8-28 所示。

对于一棵树而言，若所有结点的入度均小于等于 m，则称此树为 m 叉树。如果每个

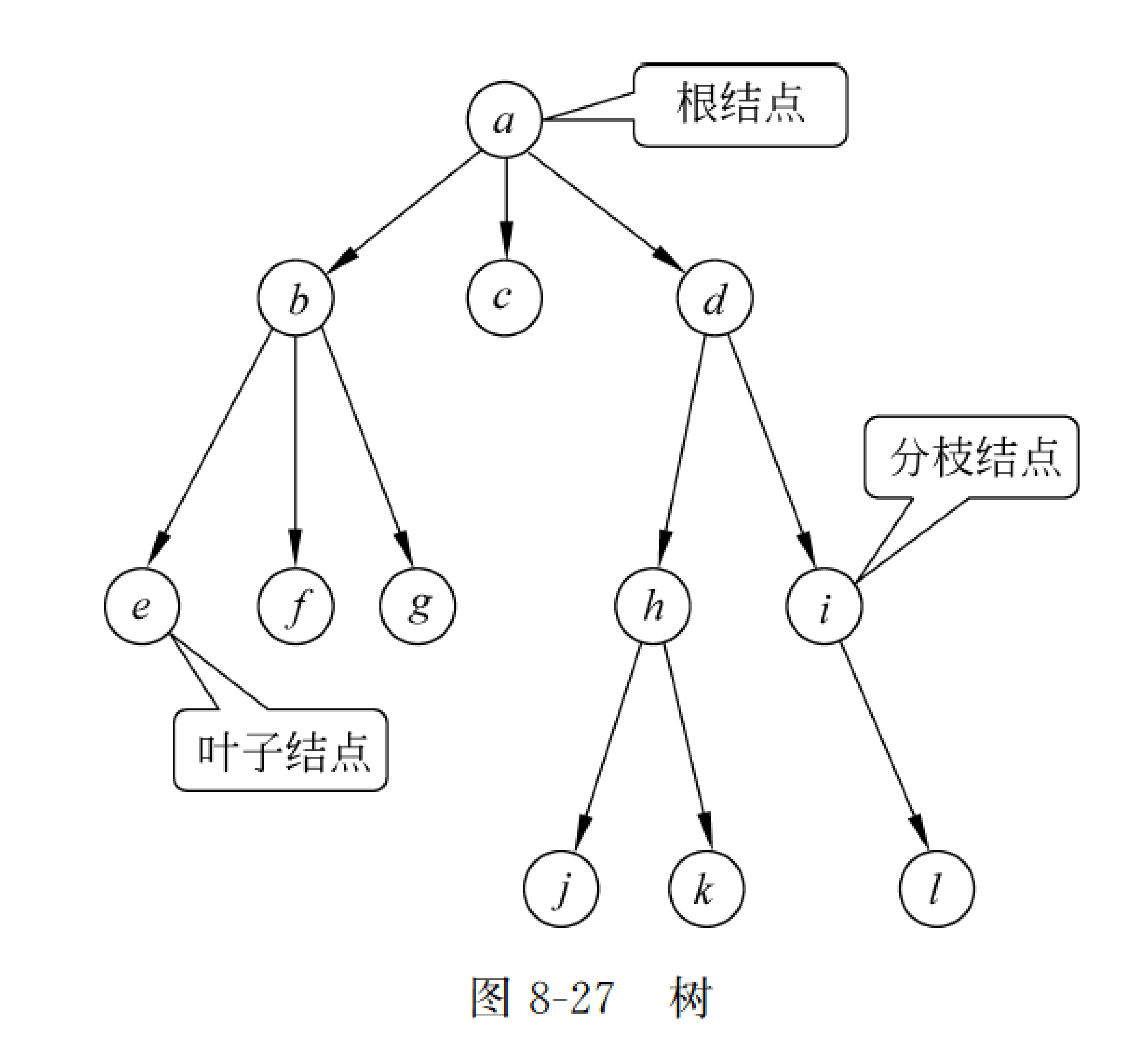


图 8-27　树

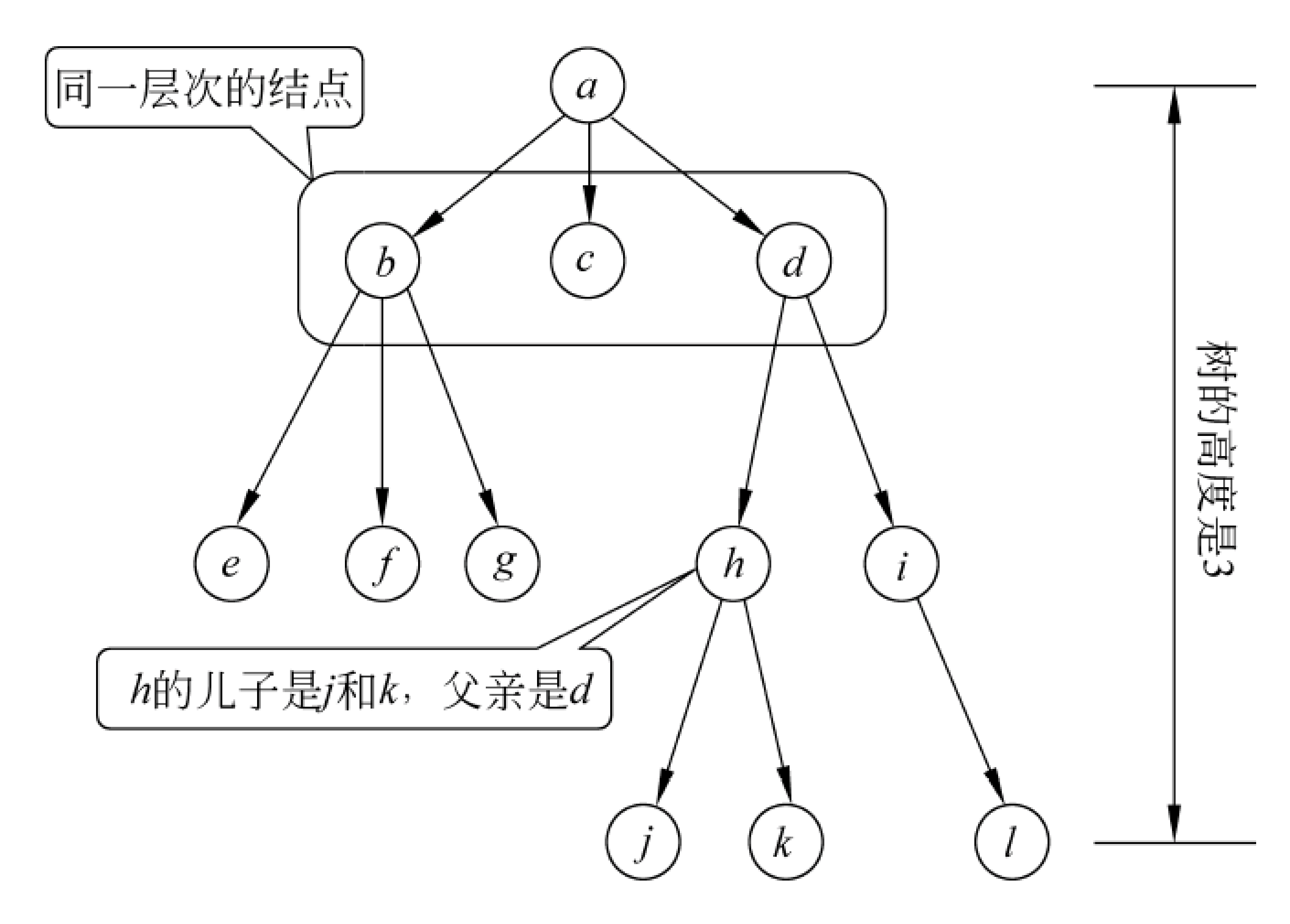


图 8-28　树的高度及层次关系

结点的入度都相等且都等于 m，则称此树为完全 m 叉树。在计算机学科经常应用的是二叉树。

8.4.4　二叉树

二叉树是每个结点最多有两个子树的树结构。通常子树被称作左子树(left subtree)和右子树(right subtree)。二叉树的每个结点至多只有两棵子树(不存在出度大于 2 的结点)，二叉树的子树有左右之分，次序不能颠倒。

二叉树具有如下性质：

- 二叉树的第 i 层至多有 2^{i-1} 个结点。
- 深度为 k 的二叉树至多有 2^{k-1} 个结点。
- 二叉树的结点个数可以为 0。
- 二叉树的结点有左、右之分。

一棵深度为 k，且有 2^{k-1} 个结点的二叉树称为满二叉树(full binary tree)。这种树的特点是每一层上的结点数都是最大结点数。

而对于深度为 K 的，有 N 个结点的二叉树，当且仅当其每一个结点都与深度为 K 的满二叉树中编号从 1 至 n 的结点一一对应时，称为完全二叉树(complete binary tree)。也就是说，若一棵二叉树至多只有最下面的两层上的结点的度数可以小于 2，并且最下层上的结点都集中在该层最左边的若干位置上，则此二叉树成为完全二叉树。具有 n 个结点的完全二叉树的深度为 $\log_2 n+1$。深度为 k 的完全二叉树，至少有 2^{k-1} 个结点，至多有 2^k-1 个结点。图 8-29(a)、(b)分别是满二叉树和完全二叉树的示例。

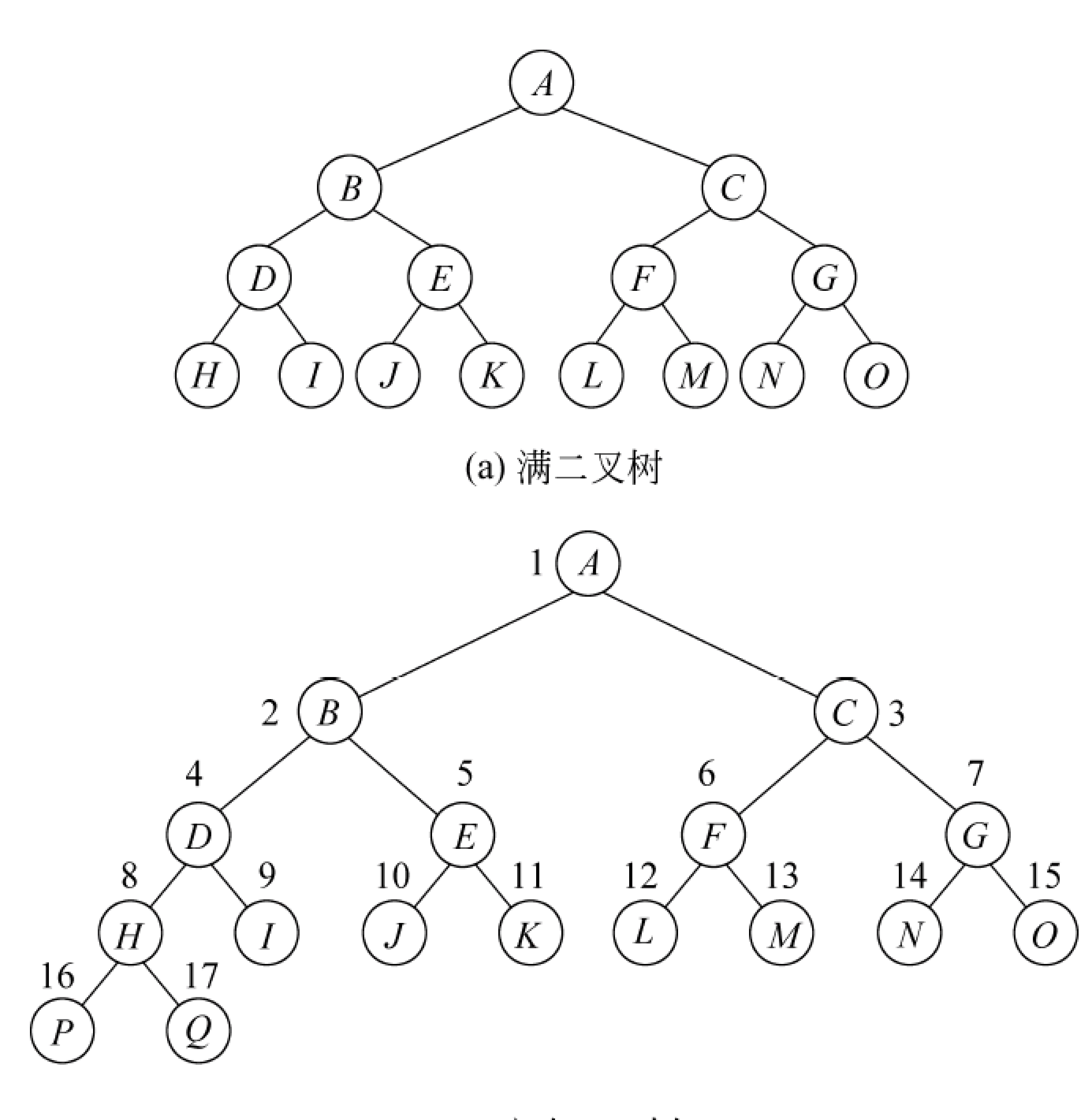


图 8-29　满二叉树和完全二叉树

8.4.5　树的遍历

树的遍历是树的一种重要的运算。所谓遍历是指对树中所有结点的信息的访问，即依次对树中每个结点访问一次且仅访问一次。树的遍历有两种方式：先根遍历，即先访问树的根结点，然后依次先根地访问根的每一棵子树；后根遍历，即依次后根地访问根的每一棵子树，最后访问根结点。

对于图 8-29 的树，采用先根遍历的方式，结点访问的次序依次为

a b e f g c d h j k i l

采用后根遍历的方式，结点访问的次序依次为

e f g b c j k h l i d a

习　　题

1. 什么是数据的线性存储结构？什么是数据的非线性存储结构？
2. 简述线性表的操作。
3. 假设电话号码簿由人名和一个电话号码组成，设计一个线性表，存储 7 个人的电话号码簿。
4. 设栈 S 中存储的是字符数据，自栈底到栈顶依次为 A、C、D。经过两次出栈操作并将 E 压入栈，此时栈中的数据是什么？
5. 使用栈，检查表达式(2＋3)＊a＊(3＋b)/(2＊(12＋8)的括号是否匹配。
6. 编写程序，输入一行文本，然后使用栈逆序显示该行文本。
7. 编写程序，用栈来判断一个字符串是否为回文(即顺读和倒读都相同的字符串)。程序忽略字符串中的大小写、空格和标点符号。
8. 设计一个队列，将整数 3、4、5 入队，打印该队列，将队列的前两个元素出队，随后将 11 和 12 入队，再次打印该队列。
9. 对于图 8-13 的循环队列，在该图的基础上，将 1、2、3、4、5 入队，并将两个元素出队后，画出队列目前的状态。
10. 将图 8-19 的带权图使用邻接矩阵的方式存储到计算机中，试写出该矩阵。(提示：邻接矩阵是一个对角线元素为 0 的对称矩阵。)

附录A 常用外设及设备驱动程序

A.1 输 入 设 备

A.1.1 键盘

键盘(keyboard)是计算机中最常用的输入设备,由按键、键盘架、编码器、键盘接口及相应控制程序等几部分组成。键盘通常有几十或上百个键,每个键相当于一个开关。一般微型机的键盘包括标准键盘(83 键、84 键)和扩展键盘(101 键、104 键)两种,如图 A-1 所示。



图 A-1 键盘

1. 键盘的构成

键盘内主要由单片机、译码器和 16 行×8 列的键开关阵列 3 部分组成。所谓单片机,就是将主机的 4 个组成部分——CPU、存储器、总线及接口集成在一片硅片上。不同性能的单片机,这 4 部分的性能、容量等有较大的差别。键盘中使用的单片机通常是 8 位字长的,内含 2KB 的只读存储器(ROM)、128B 的随机存取存储器(RAM)、2 个 8 位 I/O 接口、1 个 8 位定时/计数器以及时钟发生器等。

2. IBM PC 系列键盘的特点和分类

IBM PC 系列键盘具有两个基本特点:

(1) 按键开关均为无触点的电容开关。它是通过按键的上下动作使电容量发生变化,来检测按键的断开或接通。除电容式开关外,常见的键开关还有霍尔效应式开关和触点式开关。

(2) PC 系列键盘属于非编码键盘。键盘按照键开关的类型可分为触点式和无触点式两种;从按键材料上分则有机械触点式、薄膜式和电容式;而从功能上讲,一般又将键盘分为编码键盘和非编码键盘。

对编码键盘,当有键按下,系统可以自动检测,并能提供按键的对应键值。这种键盘

接口简单，使用方便，但价格较贵。

非编码键盘只简单提供键的行列位置(位置码或称扫描码)，而按键的识别和键值的确定等工作全靠软件完成。

PC 系列键盘不是由硬件电路输出按键所对应的 ASCII 码值，而是由单片机扫描程序识别按键的当前位置，然后向键盘接口输出该键的扫描码。按键的识别、键值的确定以及键代码存入键缓冲区等工作全部由软件完成。

目前 PC 上常用的键盘插口有 3 种，第一种是比较老式的直径 13mm 的 PC 键盘插口；第二种是最常用的直径 8mm 的 PS/2 键盘插口；第三种为 USB 接口，现在也逐渐流行起来。

A.1.2 鼠标

鼠标(mouse)也是一种常用输入设备，如图 A-2 所示，其功能与键盘的光标键相似。通过移动鼠标可以快速定位屏幕上的对象，是计算机图形界面交互的必用外设之一。

鼠标一般通过微型机中的 RS-232C 串行接口、PS/2 鼠标插口或 USB 接口与主机连接。



图 A-2 鼠标

鼠标的操作包括两种：一种是平面上的移动，另一种是按键的按下和释放。当鼠标在平面上移动时，通过机械或光学的方法把鼠标移动的距离和方向转换成脉冲信号传送给计算机，计算机鼠标驱动程序将脉冲个数转换成鼠标的水平方向和垂直方向的位移量，从而控制显示屏上光标箭头随鼠标的移动而移动。

鼠标驱动程序(mouse driver)是鼠标与应用程序之间的接口，属于系统软件，在装入内存后，入口地址存放在中断向量表中，向量码为 33H。在汇编语言程序设计中，可通过软中断指令 INT 33H 调用鼠标驱动程序中的子程序，以实现对鼠标的应用。

鼠标的分类方法很多，若按照接口类型可分为 5 类：PS/2 接口、串行接口、USB 接口、红外接口和无线接口。PS/2 鼠标用的是 6 针的小型圆形接口，串口鼠标用的是 9 针的 D 型接口，USB 鼠标使用 USB 接口，具有即插即用特性。红外接口鼠标用红外线与计算机进行数据传输，无线接口鼠标则通过无线电信号与计算机进行数据传输，这两种鼠标都没有连接线，故也称为遥控鼠标，使用起来较为灵活，不受连接线的限制。但红外接口鼠标使用时要正对着计算机，偏斜角度不能太大，而无线鼠标就没有这个限制。

按照不同的工作原理，鼠标又可以分为机械式、光电式和光机式。

最常见的鼠标是机械式鼠标，其底部有一个被橡胶包盖着的金属球，紧靠着橡胶球有两个相互垂直的转轴，在转轴上装着旋转编码器和相应电路。当鼠标器移动时，球便滚动，使两个转轴旋转，由编码器及相应电路可计算沿水平方向和垂直方向的偏移量。这种鼠标器结构简单，价格便宜，操作方便，但准确度、灵敏度差。

目前是最流行的鼠标是光机式鼠标，为光学和机械混合结构。它将两个相互垂直的

滚轴紧靠在橡胶球上，两个滚轴顶端各装有一个边缘开槽的光栅轮，光栅轮的两边分别装着发光二极管和光敏三极管。当鼠标器移动时，橡胶球滚动，带动滚轴及光栅轮转动，碰到栅轮的开槽时光线透过，遇到未开槽则不透光，从而使光敏三极管产生高低电平，形成脉冲信号。光电鼠标是在鼠标底部用一个图形识别芯片时刻监视鼠标与桌面的相对移动，根据移动情况发出位移信号。这种鼠标器传送速率快，灵敏度和准确度高，但价格较贵。

对笔记本计算机，其鼠标包括内置式和外置式两种。外置式鼠标与普通台式机鼠标完全相同。内置式鼠标则与机器合为一体，在工作原理上有指点杆式、触摸屏式和轨迹球式。

鼠标最重要的参数是分辨率。它以 dpi(像素每英寸)为单位。表示鼠标移动 1 英寸所通过的像素数。一般鼠标器的分辨率为 150～200dpi，高的可达 300～400dpi，若屏幕分辨率为 640×480 时，鼠标器只要移动 1 英寸，则对应屏幕 300～400 像素位置，基本遍历屏幕的 2/3。因此鼠标的分辨率越高，鼠标器移动距离就越短。

A.2 输出设备

输出设备用于接收或传输计算机的处理结果。最基本和最常用的就是显示器和打印机。

A.2.1 显示器

显示器的作用是将主机输出的电信号经一系列处理后转换成光信号，并最终将文字、图形显示出来。常用的显示器有阴极射线管监视器(CRT)和液晶显示器(LCD)两种。

CRT 显示器分为荫罩式和电压穿透式。目前已基本退出市场。

LCD 显示器(见图 A-3)采用的技术主要有两种：有源矩阵显示器和无源矩阵显示器。



图 A-3 LCD 显示器

有源矩阵显示器又称为薄膜晶体管液晶显示器(TFT)。它的每一个像素点都用一个薄膜晶体管来控制液晶的透光率，优点是色彩鲜艳，视角宽，图像质量高，响应速度快。但其成品率低，从而导致价格比较昂贵。

无源矩阵显示器用电阻来代替有源晶体管，制造较为容易。它和有源矩阵相比的最大优势就是价格低。其缺点是色彩饱和度较差，图像不够清晰，对比度也较低，视角较窄，响应速度慢。

LCD 显示器与 CRT 显示器相比较，其特点是外尺寸相同时可视面积更大，体积小(薄)，外形美观，图形清晰，不存在刷新频率和画面闪烁的问题，但价格比较昂贵，分辨率较低。

A.2.2 打印机

打印机也是计算机系统的标准输出设备之一(见图 A-4)。它与主机之间的数据传送方式有并行和串行两种。目前大多数打印机采用并行数据传送方式,即通过并行接口与主机连接。串行打印机则通过主机的串行口连接。

打印机的种类很多。按照打印原理,可分为击打式打印机和非击打式打印机。击打式打印机是用机械方法使打印针或字符锤击打色带,在打印纸上印出字符。非击打式打印机是通过激光、喷墨、热升华、热敏等方式将字符印在打印纸上。



图 A-4 打印机

1. 打印机的工作方式

同显示器一样,打印机在微机系统中的工作方式也可按其从主机接收的数据类型分为字符方式和图形方式。

所谓字符方式,是指主机在发送打印数据时,只传送字符的 ASCII 码,打印机根据收到的 ASCII 码从字模 ROM 中取出相应的字符点阵信息,最后用机械、光学或加热的方法打印到纸上。汉字的打印也可以在字符方式下进行,这要以打印机内部具备全部汉字字模为前提。字符方式可以获得较快的打印速度,是当前西文打印中最常用的方法,中文打印如果采用这种方式,打印机的成本就要相应提高。字符方式不能用于图形打印。

在图形方式下,主机所传送的不是字符代码,而是经过软件编辑的图形像素信息。图形方式既可以打印西文字符,也可以打印汉字或任意的图像。

在微机系统中,上述两种打印方式往往是共存的,到底使用哪一种方式要视具体情况而定。有时,用户可用键盘输入命令或通过程序中给定的指令来选择其一,有时由系统规定而不能改变。

打印机通过接口与主机相连,该接口也称为打印机控制器或适配器。它可以是一块独立的接口卡,也可以集成在主板上(现代微型机的主板上几乎无一例外地都集成了打印机的接口)。它们通过标准的 25 芯插头插座相连接。

在 CPU 与打印机进行数据传送时,首先要由接口向打印机提供“选择输入”控制信号,打印机在此信号控制下,才能接收数据及其他控制信号;同时,打印机要向接口送上有效的“打印机选中”状态信号,表示打印机已加电工作。之后,CPU 通过接口向打印机输出数据(字符)。这种输出是一个字节一个字节进行的。每一次“选通”,输出一个字节到打印机内部的缓冲存储器,直到全部数据传送完毕。许多打印机还可提供“忙”、“纸尽”等状态信号,以停止主机做相应的处理。

2. 主要性能指标

衡量打印机性能的主要指标包括以下几个方面:

(1) 分辨率。分辨率用 dpi 表示,即每英寸打印点数,它是衡量打印质量的重要指

标。不同类型的打印机其打印质量也不同。针式打印机的分辨率较低，一般为 180～360dpi，喷墨打印机分辨率一般为 300～1440dpi，激光打印机的分辨率为 300～2880dpi。

（2）打印速度。针式打印机的速度用每秒打印字符数（CPS）表示。打印速度在不同的字体和文种下差别较大。针式打印机的打印速度由于受机械运动的影响，在印刷体方式下一般不超过 100CPS，在草稿方式下可以达到 200CPS。喷墨打印机和激光打印机都属于页式打印机（即计算机输出完一整页的内容，打印机才开始打印），打印速度以每分钟打印页数（PPM）表示，一般在几 PPM 到几十 PPM 之间。

（3）汉字打印、中西文字库及打印字体。能否打印汉字是衡量打印机性能的一项重要指标。有无中文字库对打印机的打印速度影响很大。另外，打印字体也是一个影响速度的因素。目前针式打印机打印汉字字体最少为 4 种（宋体、仿宋体、楷体、黑体），打印各类英文、数字字符 5～10 种；喷墨打印机打印的西文字体有 6～8 种，中文 3 种以上；激光打印机有 3 种中文字体（宋、楷、黑）及各种英文字体。以上均指打印机自带字库的情况，若使用图形打印方式，则打印字体仅与主机支持的字体数量有关。

（4）打印缓冲存储器。打印机设置较大的缓冲存储器是为了满足高速打印和打印大型文件的需要。缓冲存储器的大小将影响打印速度。针式打印机的缓冲存储器一般为 16KB。喷墨打印机和激光打印机的缓冲存储器因在图形方式下要存储大量的图形点阵信息，并且是整页装入，其缓冲存储器较大，通常容量可达 4～16MB。

（5）打印幅面。打印幅面问题是用户直接关心的问题。对针式打印机，规格有两种：80 列和 132 列，即每行可打印 80 个或 132 个字符。对非击打式打印机，幅面一般为 A4、A3 和 B4。

（6）接口类型。打印机的接口类型主要有 3 种：并行接口、串行接口和 USB 接口。并行接口应用最广泛，所以人们往往把计算机上的并行接口俗称为打印机接口。

3. 几种常见的打印机

目前市场上常见的打印机有点阵式打印机、喷墨打印机和激光打印机 3 种。点阵式打印机现在主要用于银行、税务等部门的票据类打印，喷墨打印机和激光打印机则因其打印性能、效果等方面的优势而越来越得到更广泛的应用。

A.3 设备驱动程序

A.3.1 设备驱动程序的一般概念

设备驱动程序是对连接到计算机系统的设备进行控制驱动、以使其正常工作的一种软件。在当前流行的几乎所有的操作系统中，设备驱动程序都被认为是最核心的一类部件，处于操作系统的最深层，故要重写这些驱动程序是很困难的。

有些用户可能会遇到这样的现象：将光盘放入光盘驱动器后，计算机却找不到光驱，这是为什么呢？原因很简单，就是光驱驱动程序没有安装。在平时使用计算机时，不仅是

光驱需要安装驱动程序，还有声卡、显示卡、解压卡、网卡、modem、激光或喷墨打印机以及可移动硬盘等都需要安装驱动程序。实际上，计算机中所有的硬件都需要驱动程序，但为什么使用键盘、鼠标、软驱和硬盘就不用安装驱动程序呢？这是因为这些设备的接口规范已经标准化，不需再作任何修改就能在各种环境下使用，它们的设备驱动程序已被固化在 BIOS 中作为标准的驱动程序供操作系统或应用程序使用（也可以说它们在计算机生产过程中已经被预安装到了系统中）。

由此可知，驱动程序是通过一组预先定义好的软件接口为操作系统或应用程序提供控制硬件的能力的一组软件程序。它的好处有两点：一是由于有了驱动程序这一软件层次，使操作系统或应用程序就没有必要关心硬件设备的具体操作细节，大大降低了软件的开发难度和软件的复杂程度；二是增强了软件的兼容性，例如更换设备后，只要相应地更换驱动程序即可，而无须更换整个操作系统或应用程序。当然，如果在应用程序中不通过设备驱动程序而直接访问硬件也是可以的，但这会带来兼容性问题，也就是说硬件变化后必须重新编写全部应用程序。

由于不同的操作系统对硬件的管理、控制、使用的方式方法存在一定的差异，所以，即使是同一件硬件设备，当其在不同的操作系统中使用时，也需要各个系统中专门设计的驱动程序来支持。因此，在硬件使用前，查找硬件附带的驱动程序，查阅相关驱动程序的安装和配置方法是一项十分重要的工作。

A.3.2　硬件设备的“即插即用”概念

微软公司在开发 Windows 95 时，为解决用户对外部设备硬件参数设置的困扰而开发了一项新的功能：即插即用（Plug & Play，PnP）。这是一项用于自动处理 PC 硬件设备安装的工业标准，由 Intel 和 Microsoft 两大公司联合制定。

用户需要安装新的硬件时，往往要考虑到该设备所使用的各种资源，以避免设备之间因竞争而出现冲突（比如两个设备可能占有同样的中断号、I/O 地址等）。这是一项很麻烦的工作，而有了“即插即用”功能，就使得硬件设备的安装大大简化了，用户无须再选择如何跳线，也不必使用软件配置程序，一切都可由操作系统代替完成。但要做到“即插即用”，对安装的硬件就有一定的要求，即必须是符合 PnP 规范的，否则无法做到即插即用。即插即用是 Windows 95 及以后的操作系统最显著的特征之一，基于 Intel 体系结构的其他微机操作系统目前尚不具备该特性。

即插即用特性还需要主板具有 PnP 功能，这样在系统启动时由 BIOS 自动读取提供具有 PnP 功能的接口卡的设定参数，自动分配各项资源，并将分配后的设定参数存入主机板上的闪速存储器（flash memory），再由操作系统从主板闪存读取编排后的 PnP 界面卡相关设定参数，从而避免以往因 I/O 地址相互冲突所造成的困扰，使计算机在执行各种程序时有效地发挥系统功能。

即插即用计算机系统的具体内容包括以下几部分：

（1）支持“即插即用”的 BIOS。PnP BIOS 提供基本指令集，用于确定在系统开机自检（POST）时所需要的最基本设备，这些设备至少包括显示器、键盘、磁盘驱动器等。

(2)“即插即用”操作系统。Windows 95 是第一个支持 PnP 的操作系统，之后的 Windows 系统也都支持即插即用。

(3)“即插即用”硬件。“即插即用”硬件是指由 PnP 操作系统自动配置的一组 PC 硬件设备。PnP 也同样支持打印机、调制解调器、串行口和并行口等，基于 ISA 和 EISA 的适配卡则需要进行适当的修改。

(4)“即插即用”设备驱动程序。微软公司提供的设备驱动程序支持基本 PnP 设备，如 IDE 硬盘、CD-ROM 等。



标准 ASCII 码表及控制符号

表 B-1　标准 ASCII 码表

	列	0	1	2	3	4	5	6	7
行	高位 低位	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	Ω	n	~
F	1111	SI	US	/	?	O	_	o	DEL

注：表中的 00H～1FH 以及 7FH 为控制符，不可显示；其余的为可显示字符。

表 B-2 ASCII 码表中控制符号的定义

控制符号	英文名称	中文含义	控制符号	英文名称	中文含义
NUL	Null	空白	DLE	Data Link Escape	转义
SOH	Start Of Heading	标题开始	DC1	Device Control 1	设备控制 1
STX	Start Of Text	正文开始	DC2	Device Control 2	设备控制 2
ETX	End Of Text	正文结束	DC3	Device Control 3	设备控制 3
EOT	End Of Transmit	传输结束	DC4	Device Control 4	设备控制 4
ENQ	Enquiry	询问	NAK	Negative Acknowledge	否定
ACK	Acknowledge	承认	SYN	Synchronize	同步
BEL	Bell	响铃	ETB	End of Transmitted Block	信息组传输结束
BS	Backspace	退格	CAN	Cancel	作废
HT	Horizontal Tab	横向制表	EM	End of Medium	纸尽
LF	Line Feed	换行	SUB	Substitute	取代
VT	Vertical Tab	纵向制表	ESC	Escape	换码
FF	Form Feed	换页	FS	File Separator	文件分隔符
CR	Carriage Return	回车	GS	Group Separator	组分隔符
SO	Shift Out	移出	RS	Record Separator	记录分隔符
SI	Shift In	移入	US	Unit Separator	单元分隔符
SP	Space	空格	DEL	Delete	删除

附录C 声音和图像信息的数字化

C.1 声音信息的数字化

C.1.1 声音的基本参数

声音是通过空气传播的一种连续的波，叫声波(sound wave)。当声波到达人耳鼓膜时而使人感到的压力变化，就是声音(sound)。简言之，声音是连续变化的波形，这种连续性体现为：幅值大小是连续的，可以是实数范围内的任意值；在时间上是连续的，没有间断点。这种在时间和幅值上都连续变化的信号称为模拟信号。相应地，时间和幅值都不连续的信号称为离散信号。

要使声音信号能够被计算机处理，首先需要对其进行数字化，即，将时间和幅值均连续变化的模拟声音信号通过采样(sampling)和量化(measuring)转换为时间和幅值均不连续的离散信号，这种离散的声音信号称为数字音频信号，也就是计算机能够存储和处理的信号。

表征声音的基本参数有以下几个：

(1) 幅度(amplitude)。指声音的大小或强弱程度，幅度越大，表示声音越高。

(2) 频率(frequency)。指信号每秒钟变化的次数，用赫兹(Hz)表示。频率越高，声音听上去就越“尖锐”。低于或高于一定频率后的声音人就听不到了。

(3) 带宽(band width)。声音信号的频率范围。如高保真声音的频率范围为10～20000Hz，它的带宽约为20kHz。

(4) 亚音信号(subsonic)。频率小于20Hz的、人听不到的声音信号。人们对声音的感知不仅与声音的幅度有关，还与声音的频率有关。中频或高频中可感知的相同的音量在处于低频时需要更高的能量来传递。例如，大气压的变化周期很长，以小时或天数计算，一般人不容易感到这种气压信号的变化，更听不到这种变化。

(5) 音频信号(audio)。频率范围为20Hz～20kHz的、人能够听到的声音信号。

(6) 超音频信号(supersonic)，也称超声波(ultrasonic)，是高于20kHz的信号。

计算机中处理的声音信号主要是音频信号，包括音乐、话音、风声、雨声、鸟叫声、机器声等。音频信号的带宽(频率范围)越宽，声音的质量(音质)就越好。

C.1.2 声音信号的数字化

要使连续变化的声音信号(模拟信号)能够为计算机处理,必须要将其转变为离散(不连续)的数字信号。将时间和幅值均连续变化的模拟声音信号转换为在时间和幅值上均离散的数字信号的过程称为声音信号的数字化。这是声音信号进入计算机的第一步,数字化的主要工作就是采样和量化。

采样是指定期在某些特定的时刻对模拟信号进行测量。采样的结果是得到在时间上离散但幅值上连续变化(幅值可以是任意一个实数值)的离散时间信号(discrete time signal)。

对这种连续幅值的离散时间信号,计算机是无法进行处理的,还需要将信号幅度的取值数目加以限定,将任意实数值的幅度值转换为由有限个数值组成,使信号不仅在时间上,同时也在幅值上离散的离散幅度信号(discrete amplitude signal)。例如,设输入电压的范围是 0.0～0.7V,而它的取值仅限定在 0,0.1,0.2,…,0.7V 共 8 个值。如果采样得到的幅值是 0.123V,则近似取值为 0.1V;如果采样得到的幅度值是 0.271V,它的取值就近似为 0.3V。这种数值就称为离散数值,对幅值进行限定和近似的过程称为量化。把时间和幅值都用离散数字表示的信号就称为数字信号(digital Signal)。

对声音的数字化实质上就是采样和量化,如图 C-1 所示。只有将连续变化的模拟声音转换为离散的数字音频信号,计算机才能处理和存储。可以想象,在一个规定的时间里对模拟声音采样的次数越多,对原始信号的反映(还原)就会越准确,近似度就越好。当然,采样的次数越多,所得到的数据量就越多,需要占用的存储空间就越大。

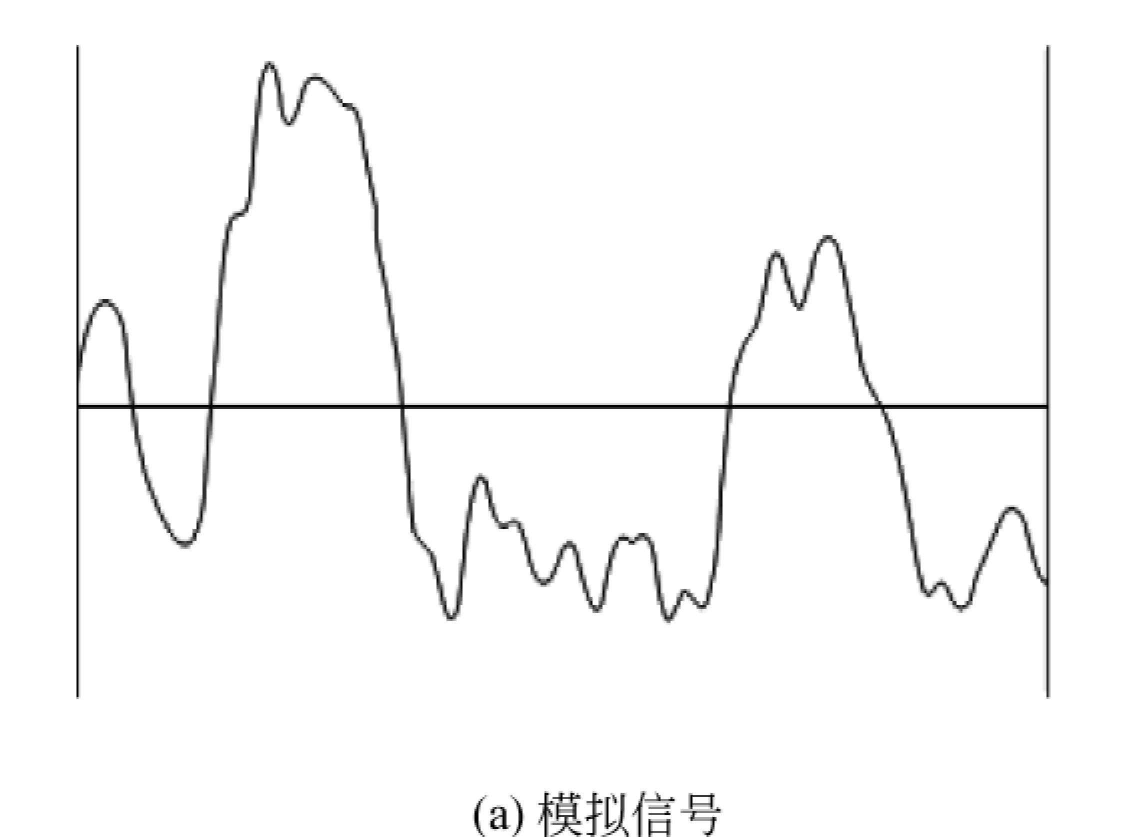
单位时间内的采样次数称为采样频率(sampling frequence)。根据奈奎斯特理论(Nyqust theory):**如果采样频率不低于信号最高频率的两倍,就能把以数字表达的声音还原成原来的声音**。例如,话音信号的最高频率为 3400Hz,采样频率至少应为 6800Hz 才能正确还原(在实际应用中,话音信号的采样频率规定为 8000Hz)。对于一般音频信号,最高频率为 20kHz,采样频率在 40kHz 以上时就能无失真地还原出原来的声音。

除了采样频率的要求外,数字化声音的不失真还原还与幅值的量化级别有关。量化级别越多,越能反映不同的声音。例如,若只用 1 位二进制码表示声音的量化级别,则只能是有声和无声两种状态;如果用 8 位二进制数表示量化级别,就可以有 256 种幅值。用以表示量化级别的二进制数的位数称为采样精度(sampling precision),也叫样本位数或位深度。对 8 位二进制数表示的声音样本,有 256 种不同的幅值,它的精度是输入信号的 1/256。

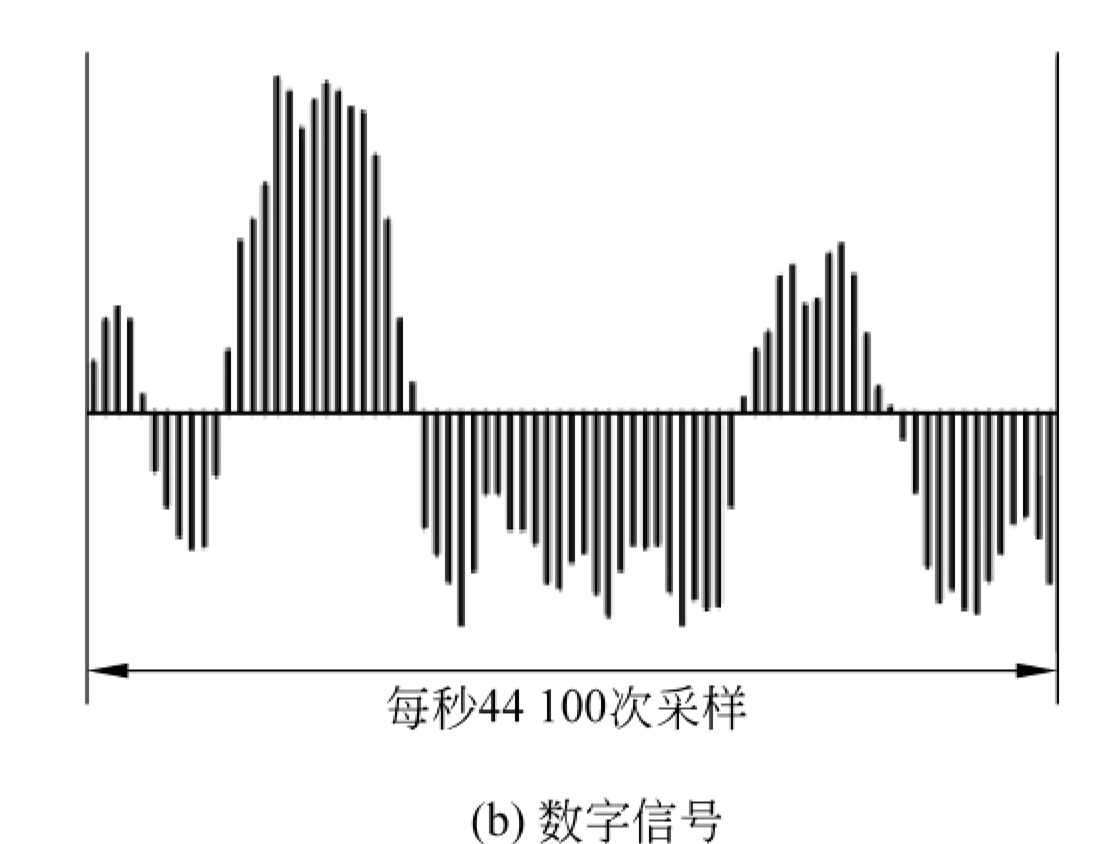
采样频率越高,样本位数越多,声音的还原性越好,质量越高,所占用的存储空间也越大。一个声音文件的大小可用下式计算:

声音文件的数据量 = 采样频率(Hz)×样本位数(b)×声道数×时间(s)

例如,对采样频率为 16kHz、样本位数为 8 位、1 分钟的单声道和双声道声音文件的数据量分别为



(a) 模拟信号



(b) 数字信号

图 C-1　声音的采样和量化

1 分钟单声道数据量＝16×8×60＝7680kb＝960KB

1 分钟双声道数据量＝16×8×60×2＝15360Kb＝1920KB

可以看出，声音文件的数据量是比较大的。为了节省存储空间，对不需高品质音效的应用程序可以使用较低的采样速率。以 Windows XP 中的录音机程序为例，如果设定语音采样速率为 8kHz，采样精度为 8 位，单声道，则文件大小只有以 44.1kHz、16 位、双声道录制的相似声音文件的 1/22。当然，这样的声音质量比较低，但在录制话音信号时（如英文阅读）已能满足要求。

C.2　图像信息的数字化

C.2.1　图像的数字化

图像是在二维空间坐标上连续变化的函数，连续图像的数字化过程是空间和幅值的离散化。空间连续坐标(x,y)的离散化称为图像的采样(image sampling)，幅值 $f(x,y)$ 的离散化称为整量。

采样是将一幅图像变换为 $f(x,y)$坐标中的一个个点，称为像素点。每一个像素点具有颜色空间中的某一种颜色（灰度值）。

采样所得到的像素点的灰度值是连续的（如同采样后的声音信号），为便于处理还必须进行整量。整量是用有限二进制数位来表示某个像素点的灰度值，所用的二进制数位越长，可以表示的灰度等级就越多。如果仅用一位二进制码表示像素点的灰度，该像素点就只有黑、白两种颜色；若用 4 位二进制码来表示，则该像素点就可以有 16 种不同的颜色（或由黑到白 16 种不同的灰度等级），相应的图像称为 16 色图像。

将一幅连续图像按一定顺序在 x 和 y 方向进行等间隔采样，就将图像变换为 $N\times N$ 个像素点组成的数组，再对这些像素点的灰度用等间隔进行整量，就得到了一幅 $N\times N$ 的数字图像(digital image)。例如，对图 2-7 所示的黑白二值图像，在横向和纵向各取 10 个点进行采样，可以得到 10×10 个数值，由于只有黑白两种颜色，故每个点的颜色（灰度）

只需用1位二进制码表示，即二级量化（如用0表示黑，用1表示白）。这样，就得到10×10个取值为0或1数据，将这些数据按采样的行列位置排列成如图2-8所示的形式，这就是一幅二值图像在计算机中的表示。

数字图像中表示每个像素颜色所使用的二进制位数称为像素深度（pixel depth）或位深度。像素深度越大，图像能表示的颜色数越多，色彩越丰富逼真，占用的存储空间越大。常见的像素深度有1位、4位、8位和24位，分别用来表示黑白图像、16色或16级灰度图像、256色（或256级灰度）图像和真彩色（2^{24}种颜色）图像。

C.2.2 图像的主要性能参数

一幅图像的采样点数称为图像分辨率（image resolution），用点的“行数×列数”表示。如果一幅图像只取一个采样点，只得到一个数据，量化后这幅数字图像就只有一种颜色。对相同尺寸的图像，采样的点数越多，图像的分辨率就越高，所得数字图像看上去就越逼真，越细腻；相反，则图像显得越粗糙。例如，图像分辨率为640×480的数码相机拍摄的照片就远比分辨率为1128×764相机拍摄的照片差。

图像分辨率是组成数字图像的像素数，在用扫描仪扫描图像时，还涉及另外一种分辨率，称为扫描分辨率（scanning resolution）。扫描分辨率用每英寸所含像素点数（dots per inch，dpi）表示，用于使不同尺寸的图像获得相同的扫描精度。

扫描分辨率和图像分辨率不同，扫描分辨率是采样时单位尺寸内采样的点数，图像分辨率是组成数字图像的像素数。例如，用200dpi来扫描一幅6in×8in的图像，得到一幅1200×1600像素的数字图像。

图像文件的大小由图像分辨率和像素深度决定。一幅位图图像文件的大小可由下式估算：

位图图像文件大小＝图像分辨率×像素深度

例如，一幅图像分辨率为640×480的真彩色图像（位深度24位）的图像文件数据量为：

$$640\times480\times24=3732800\text{b}=921600\text{B}$$

可以看出，图像的分辨率越高，样本位数越大，图像文件占用的存储空间就越大，其传输需要的时间也就越长。如果在家里从因特网下载一个640×480大小的256色位图要花费半分钟或更长时间，而下载16色同样大小的图像文件则可以减少一半的时间。

数字图像的视觉效果与图像输出设备有关，图像在屏幕上的显示尺寸称为图像的显示分辨率（display resolution）。分辨率低的图像可以以高的分辨率显示，分辨率高的图像也可以以低的分辨率显示，但只要不是以图像的正常分辨率显示图像，都会引起图像的失真。所以，使用图像时应按需要设置图像的分辨率和像素深度。

参 考 文 献

[1] 王伟. 计算机科学前沿技术. 北京：清华大学出版社，2012.

[2] 张立昂. 可计算性与计算复杂性导引. 北京：北京大学出版社，2011.

[3] 冯·诺依曼. 计算机与人脑. 陈莉，译. 南京：江苏人民出版社，2011.

[4] 李波，等. 大学计算机——信息、计算与智能. 北京：高等教育出版社，2013.

[5] 赵英良，等. 大学计算机. 4 版. 北京：清华大学出版社，2011.

[6] 吴宁，崔舒宁，等. 大学计算机基础. 北京：电子工业出版社，2013.

[7] 冯博琴，吴宁. 微型计算机原理与接口技术. 3 版. 北京：清华大学出版社，2011.

[8] 汤子瀛，哲凤屏，汤小丹. 计算机操作系统. 西安：西安电子科技大学出版社，1996.

[9] 张江. 图灵机与计算问题. http://www.doc88.com/p-984356866205.html.

[10] 谢希仁. 计算机网络. 5 版. 北京：电子工业出版社，2008.

[11] 冯博琴，陈文革. 计算机网络. 2 版. 北京：高等教育出版社，2008.

[12] 陈文革. 计算机网络技术经典实验案例集. 北京：高等教育出版社，2012.

[13] 百度百科. http://baike.baidu.com/.

[14] 维基百科. http://zh.wikipedia.org/wiki/Wikipedia.

[15] Paul Deitel，等. Visual C# 2010 大学教程. 4 版. 张思宇，等译. 北京：电子工业出版社，2011.